Fachhochschule Südwestfalen
University of Applied Sciences

Workshop “Programmieren mit C“

25. Oktober 2023

Prof. Dr. Heiner Giefers

Inhalt

1 Einleitung

Herzlich willkommen an der Fachhochschule Südwestfalen und im Workshop "Programmieren". Dieser Workshop richtet sich in erster Linie an Programmieranfänger und soll den Einstieg in die Vorlesungen des ersten Semesters etwas vereinfachen. Diejenigen von Ihnen, die schon Vorkenntnisse im Bereich der Programmierung mitbringen, sind natürlich gleichermaßen willkommen. Falls Ihnen der behandelte Stoff schon bekannt ist, wäre es nett, wenn Sie Ihre Kommilitoninnen und Kommilitonen bei der Bearbeitung der Aufgaben unterstützen und vielleicht lernen Sie ja noch etwas Neues hinzu.

Sie interessieren sich für Computer; schließlich haben Sie sich für ein Informatikstudium entschieden. Aber vielleicht haben Sie Computer bisher nur über Programme oder *Apps*, wie man neudeutsch sagt, benutzt. Im Studium werden Sie erfahren, wie man Programme konzipiert und entwickelt. Die Erstellung von Programmen nennt man *Softwareentwicklung*. Die Vorlesungen, die Sie in den ersten Semestern besuchen werden, vermitteln Ihnen das nötige Rüstzeug, um die verschiedenen Aspekte der Softwareentwicklung zu beherrschen.

Ein wichtiger Teil der Softwareentwicklung und sozusagen das Handwerkszeug eines jeden Informatikers ist das Programmieren. Beim Programmieren wird der Entwurf einer Software, also die Programmidee in Form eines strukturierten Textes aufgeschrieben. Dieser Text heißt dann auch *Quelltext* oder *Quellcode* (oder auch kurz *Code*). Wie der Code geschrieben werden muss, ist durch zwei verschiedene Aspekte vorgegeben, dem Algorithmus und der Programmiersprache.

Ein Algorithmus ist die Beschreibung einer Lösungsstrategie für ein bestimmtes Problem und meistens angegeben in allgemeiner, textueller Form. Das bedeutet, ein Algorithmus ist zunächst völlig unabhängig von einem Computerprogramm. Erst, wenn der Algorithmus in einer Programmiersprache aufgeschrieben wird, kann er vom Computer weiterverarbeitet, bzw. ausgeführt werden. Die Aufgabe von Informatikern ist es häufig, reale Problemstellungen so zu *abstrahieren*, dass sie im Computer *dargestellt* und mittels Algorithmen *gelöst* werden können.

Nehmen wir an, das Problem besteht darin, dass Sie ein Kartenspiel spielen wollen, in dem keine Joker-Karten benötigt werden, in Ihrem Kartensatz sind aber einige Joker vorhanden. Das Problem besteht nun also darin, die Joker-Karten herauszunehmen. Der folgende Algorithmus gibt eine Möglichkeit an, um dieses Problem zu lösen:

Algorithmus zum Aussortieren der Joker

Daten: Ein Spielkartensatz
Ergebnis: Der Spielkartensatz ohne Joker
Lege den Stapel der Spielkarten auf den Tisch. Lasse daneben Platz für einen neuen Stapel
Solange noch Karten auf dem Stapel sind **tue dies**
  Decke die oberste Karte des Stapels auf
  Falls Oberste Karte ist ein Joker **wahr ist**
    Lege die oberste Karte beiseite
  sonst
    Lege die oberste Karte auf den neuen Stapel.
  ende
ende
Das Ergebnis ist der neue Stapel auf dem Tisch

Wenn Sie dasselbe Problem lösen wollten, würden Sie vermutlich intuitiv so ähnlich vorgehen, wie im Algorithmus oben. Mit etwas Überlegung und Zusatzwissen könnten Sie den Algorithmus auch noch verbessern. Angenommen, Sie wissen, dass nur vier Joker im Spiel sind. Dann könnten Sie mit dem Aussortieren aufhören, sobald Sie den vierten Joker gefunden und entfernt haben. Oder Sie haben noch drei Mitspieler, die Ihnen beim Aussortieren helfen. In dem Fall könnten Sie den Kartenstapel in etwa vier gleich große Teilstapel aufteilen und alle Spieler suchen nur in Ihrem Stapel nach Jokern. So können Sie das Problem *parallel* bearbeiten, was üblicherweise deutlich schneller geht. Allerdings funktionieren dann einige Tricks, wie etwa das Mitzählen der Joker nicht ohne Weiteres. Sie müssten sich ständig mit den anderen Spielern *synchronisieren* wie viele Joker insgesamt schon gefunden wurden. Das kostet Zeit und ist vielleicht aufwändiger, als einfach den kompletten Stapel durchzusehen.

Natürlich ist das Finden von Spielkarten kein Problem, dass Sie mit einem Computer lösen würden. Aber Sie können sich leicht ähnlich gelagerte Probleme vorstellen, die per Hand nur sehr mühsam zu bearbeiten sind. Nehmen Sie statt des Kartenstapels einen Text mit abertausenden von Worten, für den Sie nun herausfinden möchten, an welchen Stellen im Text ein bestimmtes Wort oder ein bestimmter Name auftaucht. Nun müssten Sie den gesamten Text lesen, was vielleicht mehrere Stunden dauert. Ein Computer könnte Ihnen das gleiche Problem im Bruchteil einer Sekunde lösen, vorausgesetzt, die haben den passenden Algorithmus als Programm vorliegen.

Wenn das Problem nicht vorher schon einmal jemand gelöst und das Programm verfügbar gemacht hat, müssen die selbst tätig werden und ein eigenes Programm entwickeln. Den Algorithmus haben Sie sich ja bereits überlegt, nun müssen Sie ihn noch so aufschreiben,

dass der Computer ihn *versteht* und anwenden kann. Dazu verwenden Sie eine sogenannte *Programmiersprache*.

Eine Programmiersprache macht es dem Softwareentwickler sehr viel einfacher, aus seinen Algorithmen lauffähige Computerprogramme zu machen. Streng genommen müssen Sie nicht zwingend eine Programmiersprache verwenden, um ein Programm zu schreiben. Der Computer, genauer gesagt der Prozessor des Computers, führt Programme in Form von Befehlen aus. Diese Befehle heißen auch *Maschinencode*, es ist eine Art Programmiersprache, die der Prozessor direkt ausführen kann. Allerdings ist das Schreiben von Algorithmen in Form von Maschinencode sehr zeitaufwändig und fehleranfällig. Mit Programmiersprachen können Algorithmen viel abstrakter, d. h. für Menschen verständlicher, formuliert werden. Damit wird die Programmentwicklung deutlich vereinfacht.

Nehmen wir nochmals Algorithmus zum Aussortieren von Spielkarten als Beispiel. In der textuellen Beschreibung finden wir einen Ausdruck **Solange**, gefolgt von der **Bedingung** *„noch Karten auf dem Stapel sind“*. Das bedeutet, solange die Bedingung erfüllt ist, soll der Block nach der Zeile (erkennbar durch die Einrückung und den senkrechten Strich) wiederholt werden. Wir werden sehen, dass wir diesen Text fast 1:1 in einen Programmtext überführen können. Das Programmierkonstrukt[1], das wir dazu verwenden, heißt **Schleife** und wird von den allermeisten Programmiersprachen unterstützt[2]. Ebenso verhält es sich mit dem Ausdruck nach **Falls**. In der Programmierung nennt man dieses Konstrukt *Verzweigung*.

Neben den Ausdrücken, mit denen der Ablauf des Programms gesteuert wird, sind auch die Daten von zentraler Bedeutung. Als wichtigstes Datenelement verwendet unser Algorithmus den Kartenstapel. Neben dem Ausgangsstapel gibt es noch einen zweiten Stapel, den wir nach-und-nach durch das Herausnehmen der Joker neu bilden. Die Kartenstapel sind zusammenhängende Gebilde aus Elementen gleichen Typs, den Spielkarten. In der Programmierung bezeichnet man eine solche Zusammensetzung als *Feld* oder auch *Array*.

Wenn wir nun ein einzelnes Element des Feldes betrachten, stoßen wir auf eine Fragestellung, die in der Softwareentwicklung typisch ist. Wie soll man ein reales Objekt aus dem Zusammenhang des Problems, also in unserem Fall eine Spielkarte, im Computer darstellen? Programmiersprachen sind zumeist dazu da, Programme für beliebige Probleme zu entwickeln. Dazu besitzen Sie Programmierelemente,

[1] Als *Konstruk* bezeichne ich im Folgenden einzelne *Bausteine* von Programmiersprachen, mit denen ein bestimmtes Konzept umgesetzt werden kann.

[2] Im Gegensatz dazu besitzen die meisten Prozessoren keine Befehle, mit denen man direkt eine Schleife programmieren könnte. Stattdessen verwenden Prozessoren sogenannte *Sprungbefehle* mit denen beliebige Stellen im Programm *angesprungen* werden können.

die mit sehr generellen Datentypen umgehen können. Dies sind in der Regel

- Ganze Zahlen
- Kommazahlen
- Wahrheitswerte (wahr/falsch bzw. englisch *true/false*)
- Buchstaben/Zeichen

Eine Spielkarte ist aber ein sehr spezielles Objekt und es ergibt nur wenig Sinn, die oben genannten Typen durch einen weiteren Typ *Spielkarte* zu ergänzen. Die ProgrammiererIn muss sich also überlegen, wie man eine Spielkarte mittels der eingebauten, vorhandenen Datentypen *darstellt*. Eine Möglichkeit das zu tun, ist alle Spielkarten durchzunummerieren und jeder Karte eine eindeutige ganze Zahl zuzuordnen. Solange man die Art der Nummerierung kennt, kann man jeder gültigen Zahl dann wieder den Wert der Karte zuordnen. Wir werden im Verlauf der Vorlesung *Programmierung mit C++ 1* noch sehen, dass man auch neue Typen aus mehreren bestehenden Datentypen zusammensetzen kann. Ein Typ, der Spielkarten repräsentiert, könnte z. B. aus zwei Komponenten zusammengesetzt sein: Der Wert der Karte (2, 3, …Bube, Dame König, Ass) als ganze Zahl und die Farbe (Karo, Herz, Pik, Kreuz) als Zeichen (K, H, P, X[3]). Dieses Programmierkonstrukt nennt sich **Struktur** und ist von zentraler Bedeutung für die Objektorientierte Programmierung, die Sie im zweiten Semester in der Veranstaltung *Programmierung mit C++ 2* näher kennenlernen werden.

Im weiteren Verlauf dieses Workshopes soll es darum gehen, erste Programmiererfahrungen in der Programmiersprache **C** zu sammeln. Wie Sie im nächsten Kapitel sehen werden, gibt es eine Vielzahl verschiedener Programmiersprachen. Dass wir hier die Sprache C verwenden, hat mehrere Gründe: Zunächst ist C eine frühe und sehr weit verbreitete Programmiersprache. Viele andere Sprachen verwenden die Programmierkonstrukte aus C in nahezu identischer Form. Darüber hinaus ist der Sprachumfang von C relativ gering, was es möglich macht, die Sprache in wenigen Wochen *komplett* zu erlernen. Ein weiterer, sehr pragmatischer Grund, warum wir C im Workshop einsetzen ist, dass die Sprache auch in der Vorlesung *Programmierung mit C++ 1* des ersten Semesters behandelt wird. Mit den Kenntnissen aus dem Workshop werden Sie es vermutlich leichter haben, in die Vorlesung einzusteigen.

[3] Da Karo und Kreuz beide mit einem *K* beginnen, verwenden wir für Kreuz statt des K's den Buchstaben X.

2 Programmiersprachen

Seit der Entwicklung der ersten modernen Computer in den 1950er Jahren, wurden tausende von Programmiersprachen entworfen und es kommen weiterhin immer wieder neue Sprachen hinzu. Der Großteil dieser Sprachen hat allerdings keine große Relevanz für die Mehrheit der Softwareentwickler.

Es gibt einige Programmiersprachen, die für ganz spezielle Problemklassen entwickelt worden sind, die sogenannten *domänenspezifischen Sprachen*. Beispiel sind etwa Sprachen für Datenbankabfragen, zum Erstellen von Grafiken oder zum Entwurf von Webseiten. Diese Sprachen sind sehr sinnvoll, denn Sie erhöhen i. d. R. die Produktivität für ihr entsprechendes Einsatzgebiet und da sie speziell auf eine Problemklasse angepasst sind, sind sie meistens auch recht einfach zu erlernen. Allerdings sind domänenspezifische Sprachen normalerweise nicht sehr brauchbar, um allgemeine Probleme zu lösen.

Daher gibt es auch noch eine Reihe von sogenannten *Universalsprachen* (engl. *General Purpose Language*) , die entwickelt wurden, um für möglichst viele Anwendungsfälle benutzbar zu sein. Im Informatikstudium in Iserlohn werden Sie im Rahmen des Studiums mindesten zwei Universalsprachen erlernen. Vielleicht stellen Sie sich die Frage, warum Sie mehrere Sprachen lernen müssen, wenn man doch mit einer schon „quasi alles“ programmieren kann? Die verschieden Sprachen haben aber ihre speziellen charakteristischen Eigenschaften und je nachdem was für ein Problem Sie lösen wollen, ist vielleicht die eine oder andere Sprache besser geeignet. Als Informatiker werden Sie im Studium und später auch im Berufsleben auf immer wieder neue Sprachen treffen, die Sie für Ihre jeweilige Arbeit erlernen und einsetzen werden.

Aber es gibt auch eine gute Nachricht: Sobald Sie eine Programmiersprache beherrschen, wird es Ihnen wahrscheinlich sehr viel einfacher fallen, eine neue Sprache zu erlernen. Dies gilt vor allem, wenn die neue Sprache einem Programmierstil (genauer gesagt, einem *Programmierparadigma*) folgt, das Sie schon aus einer anderen Sprache kennen. Der Begriff *Programmierparadigma* gibt an, nach welchem grundlegenden Stil Programme in einer bestimmten Klasse von Programmiersprachen entworfen werden.

Ein Programmierparadigma, das von sehr vielen Sprachen, darunter C/C++ und Java, unterstützt wird, ist die *Imperative Programmierung*. Bei der imperativen Programmierung gibt der Entwickler an, welche *Anweisungen* der Computer in welcher Reihenfolge ausführen soll. Der Programmierstil orientiert sich dabei stark an den Möglichkeiten des Prozessors, der ja selbst Maschinenbefehle nach-und-nach abarbeitet. Da der Ablauf des Programms im Vordergrund steht, las-

sen sich die meisten Algorithmen sehr gut mit einem imperativen Programmierstil umsetzen.

Im Gegensatz zur imperativen Programmierung steht beim *Deklarativen Programmierparadigma* nicht der Ablauf des Programms im Vordergrund. Vielmehr gibt der Entwickler an, welche Funktionen (*Funktionale Programmierung*) oder Fakten und Regeln (*Logische Programmierung*) im Programm gelten. Das Programm wendet dann diese Funktionen oder Ableitungsregeln an, um für eine Eingabe die entsprechende Ausgabe zu berechnen. Viele moderne Sprachen verstehen sich als sogenannte *Multiparadigmensprache*. Das bedeutet, sie besitzen Sprachkonstrukte, die verschiedenen Paradigmen folgt.

Abbildung 2.1 zeigt ein die aktuell verbreitetsten Programmiersprachen laut dem TIOBE-Index [TIO20]. Einen Großteil dieser Sprachen werden Sie im Verlauf des Bachelorstudiums kennenlernen. C/C++ werden Sie in den Veranstaltungen *Programmierung mit C++ 1 und 2* von Grund auf lernen, im dritten und vierten Semester folgen die Veranstaltungen *Java Programmierung 1 und 2*. PHP und evtl. JavaScript werden im Modul *Internettechnologien* behandelt und Python ist Kern der Vorlesung *Skriptsprachen*. Im Modul *Datenbanken* lernen Sie die Sprache SQL kennen.

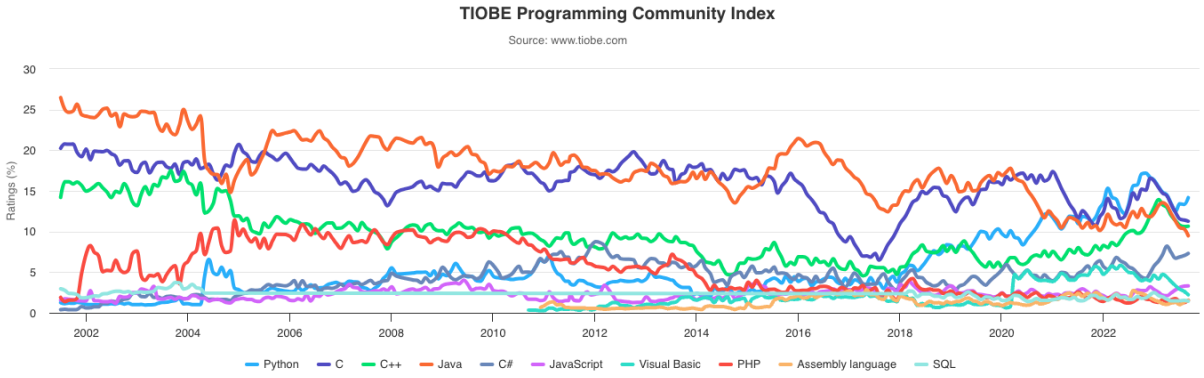

Abbildung 2.1: Ranking der beliebtesten Programmiersprachen nach dem TIOBE-Index

Im Workshop, wie auch im ersten Semester starten wir mit der Sprache C. Die Programmiersprache C entstand Anfang der 70er-Jahren und ist bis heute die wichtigste hardwarenahe Programmiersprache. Der Begriff „hardwarenahe“ bedeutet, dass die Sprache möglichst gut (d. h. vor allem *schnell* und *sicher*) mit dem Prozessor und der restlichen Hardware des Computers zusammenarbeitet. Alle gängigen Betriebssysteme (Windows, Linux, MacOS) basieren zu einem bedeutenden Teil auf C. Ein zweites wichtiges Feld der hardwarenahen Programmierung sind eingebettete Systeme. Dabei handelt es sich um Computer, die nicht wie ein klassischer PC oder Notebook aussehen, sondern in anderen technischen Geräten wie etwa Fahrzeugen, In-

dustrierobotern, Unterhaltungselektronik oder Messgeräten verbaut sind.

Allerdings hat das Erlernen von C als erste Programmiersprache auch einige Nachteile, die hier nicht verschwiegen werden sollten. Der Sprachumfang von C ist vergleichsweise gering. Das ist einerseits gut, weil die Sprache damit einfacher zu erlernen ist. Der Nachteil ist, dass der Entwickler damit nicht den Funktionsumfang zur Verfügung hat, der es erlauben würde, komplexere Programme mit wenigen Zeilen Programmcode zu entwickeln. Vielleicht werden Sie die Problemstellungen die wir hier im Workshop und auch in der Vorlesung *C++ 1* behandeln werden als etwas *langweilig* empfinden. Sobald Sie aber die Grundlagen der Programmierung beherrschen, werden Sie in der Lage sein auch größere Software zu entwickeln. Für die Sprache C, vor allem aber auch für C++ werden viele zusätzlichen Funktionen (z. B. zur Verarbeitung von Texten oder zur Programmierung grafischer Bedienoberflächen) werden über sogenannte **Bibliotheken** bereitgestellt. Wenn Sie diese Bibliotheken in Ihren Programmen verwenden, können Sie von vorgefertigten Modulen und Funktionen profitieren.

3 Die Programmiersprache C

C ist eine kompilierte Programmiersprache. Das bedeutet, dass Sie den Quelltext Ihres Programms, bevor Sie es auf einem Computer ausführen können, zunächst in ein Maschinenprogramm übersetzen müssen. Dieses Übersetzen übernimmt ein weiteres Programm für Sie, der sogenannte **Compiler**. Nach dem Übersetzen – oder, wie wir es ab jetzt nennen werden, dem Kompilieren – kann das Programm auf dem Computer gestartet werden.

Ein kompiliertes Programm ist auf dem Computer selbständig lauffähig. Das bedeutet, es kommt, bis auf das Betriebssystem und eventuell einige Bibliotheken, ohne weitere Programme aus und Sie können es nach Belieben auf dem Computer neu starten. Erst wenn Sie den Quelltext Ihres Programms ändern, müssen Sie erneut kompilieren, damit Ihre Änderungen in das ausführbare Programm übernommen werden.

Den Quelltext zu kompilieren ist nicht bei allen Programmiersprachen erforderlich. Bei vielen modernen Sprachen wird der Quelltext quasi direkt ausgeführt. Da der Prozessor den Quelltext des Programms aber nicht direkt versteht, ist dazu ein weiteres Programm nötig, der sogenannte **Interpreter**. Jedes Mal, wenn Sie Ihr Programm ausführen wollen, müssen Sie aber auch gleichzeitig den Interpreter starten, damit dieser Ihr Programm *interpretieren kann*. Daher kann man sagen, dass das Programmieren mit einer interpretierten Sprache etwas einfacher geht (weil das Kompilieren wegfällt), die fertigen Programme aber nicht ganz so „schnell“ sind.

3.1 Das erste C-Programm

Jedes C-Programm folgt einem gewissen Aufbau, dass Sie bei der Entwicklung einhalten müssen. Quellcode 3.1 zeigt ein einfaches Programm in C. Das Beispiel des “Hello World“-Programms stammt aus dem Buch *The C Programming Language*, das die Entwickler der Programmiersprache als Dokumentation erstmals 1978 veröffentlichten [KR88]. Seither ist es unter Programmierern üblich, beim Erlernen einer neuen Programmiersprache mit einem “Hello World“-Programm zu beginnen. Damit kann man zwei wichtige Aspekte überprüfen, die für das weitere Programmieren unerlässlich sind. Ist man in der Lage dieses einfache Programm auf seinem Computer zu kompilieren, so können im Normalfall auch alle weiteren gültigen Programme auf dem Computer übersetzt werden. So testet die EntwicklerIn die *Programmierumgebung* auf seinem Computer. Wenn man das “Hello World“-Programm erfolgreich startet, sollte es die Worte “hello, world“ auf den Bildschirm drucken. Ist dies erfolgreich, so hat man eine Möglichkeit innerhalb seines Programms **Ausgaben** zu produzieren. Also obwohl dieses kleine Programm recht wenig tut, kann man damit grundlegende Funktionen überprüfen und hat danach eine gute Ausgangslage um komplexere Programme zu entwickeln. Ihre erste Aufgabe im Workshop wird es also sein, ein “Hello World“-Programm auf dem Laborrechner oder Ihrem privaten Notebook zum Laufen zu bringen.

Quellcode 3.1: “Hello World“-Programm in C

```
1 #include <stdio.h>
2 int main() {
3     printf("hello, world!");
4     return 0;
5 }
```

3.2 Funktionen

Jedes C -Programm muss, unabhängig von seiner Größe, ein Hauptprogramm enthalten. Ein C -Hauptprogramm hat immer die folgende Grundform:

```
int main () {
}
```

Wenn der Compiler das Programm übersetzt, sucht er nach einem Startpunkt, bei dem das Programm anfangen soll. Dieser Startpunkt ist in C (wie auch in vielen anderen Programmiersprachen) eine **Funktion** mit dem Namen `main`. Für den Moment können Sie sich

eine Funktion als eine Art Unterprogramm vorstellen. Eine Funktion kann selbst wieder andere Funktionen aufrufen und daher ist es wichtig zu wissen, bei welcher Funktion das Programm beginnen soll.

Hinter dem Namen der Funktion `main` steht ein Paar runder Klammern. Genaugenommen sind es diese Klammern, die erkennen lassen, dass es sich bei einem Namen (also hier `main`) um eine Funktion handelt. Im Beispiel sind diese Klammern noch leer, wir werden später sehen, dass wir innerhalb dieser Klammern der Funktion Daten/-Werte (die sogenannten Argumente) übergeben können. Auf die runden folgt nun ein Paar geschweifte Klammern. Darin der sogenannte *Funktionskörper*, also dass, was die Funktion *tun* soll.

Im Quellcode 3.1 tut die `main`-Funktion genau zwei Dinge, sie ruft eine weitere Funktion mit dem Namen `printf` auf, danach führt sie die Anweisung `return 0` aus. Bleiben wir kurz bei der letzten Anweisung. Ein `return` verursacht immer, dass die Funktion beendet und verlassen wird. Ruft man `return` in der `main`-Funktion auf, beendet man damit das Programm. Dabei kann jede Funktion ein Ergebnis zurückgeben. Der Wert dieser Rückgabe wird nach dem Schlüsselwort `return` angegeben. In unserem Beispiel gibt die `main`-Funktion (und damit das gesamte Programm) den Wert 0 zurück. Damit wird dem Betriebssystem des Computers mitgeteilt, dass das Programm ohne Fehler beendet wurde[4]. Das Wort vor dem Namen der Funktion, also im Beispiel **int**, gibt an, von welchem Datentyp die Rückgabe der Funktion sein soll. Wir werden später ausführlicher über Datentypen sprechen. Sie können sich aber jetzt schon merken, dass **int** für *integer* steht, was auf deutsch soviel viel wie *ganzzahlig* bedeutet. Integerwerte sind also ganze Zahlen in einem bestimmten Wertebereich[5].

Das Prinzip der Funktion kennen Sie sicher aus der Mathematik. Wenn Sie eine mathematische Funktion $f : \mathbb{N} \to \mathbb{N}, x \mapsto x^2$ definiert haben, können Sie mit f das Quadrat jeder beliebigen ganzen Zahl x ausrechnen[6]. Also z.B. ergibt $f(3)$ den Wert 9. Hierbei sind f der Name der Funktion, $\mathbb{N}$ der Definitions- und Wertebereich der Funktion, x der Parameter, 3 das Argument und 9 die Rückgabe des Funktionsaufrufs $f(3)$. Quellcode 3.2 zeigt eine C-Funktion *quadrat*, die der mathematischen Funktion f entspricht.

Quellcode 3.2: Funktion zur Berechnung des Quadrats einer Integer-Zahl

[4] Die Rückgabe des Hauptprogramms ist ein Spezialfall und beruht auf einer Konvention des Betriebssystems. Eine Rückgabe von 0 gilt üblicherweise als *kein Fehler*, Werte ungleich 0 werden als Fehlercode interpretiert.

[5] Wenn der Compiler Integer als 32-bit Werte ablegt, ist der Wertebereich von –2.147.483.648 bis –2.147.483.647.

[6] Eine Kursschreibweise, die den Definitions- und Wertebereich der Funktion außer Acht lässt lautet $f(x) = x^2$

```
1 int quadrat(int x) {
2     return x*x;
3 }
```

Zurück zum Quellcode 3.1. Bei der Beschreibung der `main`-Funktion haben wir die erste Zeile übersprungen. Die Zeile `#include<stdio.h>` bedeutet, dass wir eine bestehende Datei von der Festplatte des Computers in unser Programm *aufnehmen* bzw. inkludieren (engl. *to include*). Dabei handelt es sich um die Datei *stdio.h*. das steht kurz für *standard input/output library functions*. An der Endung **.h** können Sie erkennen, dass die Datei eine sogenannte Header-Datei ist. In diesen Dateien stehen Informationen, die von andern Quellcode-Dateien genutzt werden sollen. In *stdio.h* stehen u. a., die Namen und Parameter von Funktion zur Ein- und Ausgabe auf der Kommandozeile. Darunter ist auch die Funktion `printf`, die unser Hauptprogramm verwendet. *printf* steht für **print f**ormatted , also *drucke formatiert*, und ist eine Standard-Funktion, die in fast jedem C-Programm benutzt wird. Das erste Argument der Funktion ist eine Zeichenkette (engl. *string*), die im Programmfenster ausgegeben werden soll. Wir werden später sehen, wie wir vom Programm berechnete Werte in diese Zeichenkette einbetten können. Neben der Ausgabe in eine Datei ist das Drucken auf der *Standardausgabe* der wichtigste Weg um Informationen vom Programm an den Benutzer zu geben.

Eine Sache, die Ihnen im Beispiel 3.1 vielleicht schon aufgefallen sein wird ist, dass in C einzelne Anweisungen immer mit einem Semikolon (`;`) abgeschlossen werden. Das Vergessen eines Semikolons ist ein typischer Fehler, der auch erfahrene Programmierer immer mal wieder passiert. Glücklicherweise fällt dieser Fehler beim Übersetzten schnell auf, sodass er leicht zu korrigieren ist. Was *übersetzen* an dieser Stelle bedeutet und wie Sie Ihr erstes Programm übersetzen können, sehen Sie im nächsten Abschnitt.

3.3 C-Programme kompilieren

Es gibt mehrere Arten, wie Sie C-Programme entwickeln können. Als Software-Programmierer werden Sie sich vermutlich eine sogenannte IDE (engl *integrated development environment*) installieren. Das ist ein Programm, dass Sie dabei unterstützt, korrekten und gut formatierten Quellcode zu schreiben und zu übersetzen. Sobald Sie eine IDE installiert haben, können Sie C-Quellcode *mit einem Klick* kompilieren und ausführen. IDEs sind allerdings häufig recht komplex und bieten viele Einstellungsmöglichkeiten, die Sie am Anfang kaum benutzen werden.

Als Informatiker sollten Sie auch einen weiteren Weg kennen, um Programme zu übersetzen und auszuführen; nämlich über die so-

genannte *Kommandozeile* Ihres Computers. Die meisten von Ihnen wissen sicherlich, dass Sie Ihren PC – ganz gleich, ob Sie Windows, Linux oder MacOS verwenden – nicht nur über grafische Menüs, sondern auch über die Kommandozeile (auch *Shell* genannt) steuern können. Auf Windows-Systemen erreichen Sie die Kommandozeile über das Programm *cmd.exe*. Linux Systeme starten entweder direkt in die Kommandozeile oder aber Sie öffnen eine Shell, indem sie ein *Terminal*-Fenster öffnen.

Über die Kommandozeile können Sie nun zwei Schritte manuell erledigen. Sie rufen den Compiler auf und teilen ihm den Namen der Datei mit, in der der Quelltext Ihres Programms steht. Falls der Compiler keinen Fehler in Ihrem Quelltext findet, übersetzt er den Quellcode in ein ausführbares Programm. Dieses Programm können Sie dann über die Kommandozeile direkt aufrufen.

Der unter Linux am häufigsten verwendete C-Compiler ist **gcc** (das steht für *GNU Compiler Collection*). Quellcode 3.3 zeigt, wie ein C-Programm mit dem gcc-Compiler auf der Linux-Shell kompiliert und ausgeführt werden kann. Nachdem man die Shell gestartet hat, wechselt man zuerst mit dem Kommando **cd** (für engl. *change directory*) in das Verzeichnis, in dem die Quelltext-Datei gespeichert ist. Im Beispiel ist das der Ordner `dev/C++1`Danach wird der gcc-Compiler mit der Quelltextdatei `hello.c`aufgerufen. Der Parameter `-o hello`gibt an, dass das erzeugte Programm den Namen `hello`tragen soll. Wenn das Kompilieren erfolgreich war, kann das Programm direkt in der Kommandozeile mit `./hello`aufgerufen werden. Die mit *printf* erzeugte Ausgabe sollte nun unterhalb des Programmaufrufs ausgegeben werden.

Übrigens sollten Sie in der Linux Shell immer darauf achten, beim Aufrufen die Zeichen `./` vor den Programmnamen zu setzen. Damit geben Sie an, dass das Programm (hier `hello`) im aktuellen Verzeichnis gesucht werden soll. Falls Sie das vergessen, wird die Shell Ihr Programm nicht finden, auch wenn es sich im aktuellen Ordner befindet.

Quellcode 3.3: Kompilieren und Ausführen des "Hello World"-Programms

```
1 hgi@HANK:~$ cd dev/C++1/
2 hgi@HANK:~/dev/C++1$ gcc hello.c -o hello
3 hgi@HANK:~/dev/C++1$ ./hello
4 hello, world!
```

3.4 Ein etwas umfangreicheres Programm

In diesem Abschnitt wollen wir nun unser minimales "Hello World"-Programm etwas ergänzen. Der Quellcode 3.4 zeigt das erweiterte

Programm mit einigen neuen Konstrukten. In den Zeilen 3 bis 5 steht ein sogenannter Blockkommentar. Blockkommentare beginnen mit den Zeichen /∗ und enden mit ∗/, dazwischen können beliebige Zeichen (unter anderem auch ∗) stehen. Neben den Blockkommentaren gibt es Zeilenkommentare, die durch die Zeichenfolge // eingeleitet werden und bis zum Ende der Zeile gehen.

Kommentare haben keinerlei Auswirkungen auf die Funktion Ihres Programms, der Compiler wird Sie beim Übersetzen einfach ignorieren. Trotzdem zählen Kommentare zu den wichtigsten Elementen im Quelltext eines Programms. Der Grund dafür ist, dass Kommentare den Quelltext eines Programms wesentlich verständlicher machen. Der Entwickler kann über Kommentare direkt an bestimmte Code-Stellen schreiben, was er sich bei der entsprechenden Stelle gedacht hat. Wir werden sehen, dass Entwickler an vielen Stellen im Programm Namen vergeben müssen. Diese Namen sollten einerseits selbsterklärend sein, andererseits aber auch nicht zu lang gewählt werden, da ansonsten der Code unübersichtlich wird. Durch Kommentare kann der Programmierer an einer Stelle ausführlich beschreiben, was mit dem Namen gemeint ist. Das hilft anderen Personen und auch dem Entwickler selbst, den Code später zu verstehen und nachvollziehen zu können.

Quellcode 3.4: "Hello World" – die Zweite

```
1  #include <stdio.h>
2
3  /*
4  * Ein etwas umfangreicheres Programm
5  */
6  int main() {
7      int version;     // Versionsnummer des Programms
8      version=2;     // Hier wird version initialisiert
9      printf("hello, world. Die %d.\n", version);
10     return 0;
11 }
12 /* Ende des Programms */
```

Die zweite Neuerung in Programm 3.4 ist die Einführung einer **Variablen**. In Zeile 7 wird die Variable `version`definiert und in Zeile 8 mit dem Wert 2 initialisiert. Was Datentypen sind und welche Typen die Sprache C beinhaltet, werden wir im nächsten Kapitel behandel.

Die `printf`-Funktion in Zeile 9 sieht so ähnlich aus, wie im vorherigen Beispiel. Allerdings hat dieser Aufruf nun 2 Parameter, statt nur einem. Neben der in Anführungszeichen stehenden Zeichenkette wird der `printf`-Funktion im Quellcode 3.4 zusätzlich der Wert der Variablen `version`mitgegeben. In diesem Fall ist der Wert 2. Die `printf`-Funktion wird diesen Wert an einer bestimmten Stelle in der Zeichenkette anzeigen, und zwar genau dort, wo ein sogenanntes **Formatierungszeichen** angegeben ist. Formatierungszeichen begin-

nen immer mit dem Zeichen **%**; danach stehen andere Zeichen, die angeben, *wie* ein bestimmter Wert ausgegeben werden soll. Wenn Sie eine Variable vom Typ `int` „ganz normal“ ausgeben wollen, verwenden Sie das Formatierungszeichen `%d`. Das *d* steht hierbei für *dezimal*. Das Zeichen `\n` hat übrigens auch eine spezielle Bedeutung, damit wir ein Zeilenumbruch erzeugt. Das Zeichen, das als nächstes ausgegeben wird, erscheint also in einer neuen Zeile innerhalb der Eingabeaufforderung.

3.5 Lernkontrollfragen

a. Was ist der Unterschied zwischen einem kompilierten und einem interpretierten Programm?
b. Wie ist eine C-Funktion aufgebaut?
c. Kann eine Funktion mehrere `return`-Anweisungen ausführen?
d. Welche Aufgabe haben Kommentare?
e. Wie viele Parameter hat die **`printf`**Funktion?
f. Was ist ein Formatierungszeichen?
g. Schreiben Sie ein C-Programm, dass Sie persönlich (also mit Namen) begrüßt, wenn Sie es aufrufen. Verwenden Sie dazu die Programmierumgebung aus Abschnitt **??**.

4 Schnellkurs C

In diesem Kapitel werden wir die wichtigsten Sprachelemente von C relativ kurz behandeln. Natürlich kann man die komplette Sprache nicht auf ein paar Seiten beschreiben, aber der hier vorgestellte Ausschnitt sollte genügen, um erste Programme selbständig schreiben zu können. Wir werden die Feinheiten der Sprache dann noch ausgiebig in der Vorlesung besprechen. Ich rate Ihnen auch dringend, sich zusätzliche Lehrbücher zum Thema aus der Bibliothek auszuleihen. Schauen Sie doch einfach mal unter `https://kai.fh-swf.de` nach, welche Bücher in der Bibliothek vorrätig sind. Wenn Sie den Suchfilter *Verfügbarkeit* auf *online* stellen, sehen Sie nur diejenigen Bücher, die Sie online lesen oder sogar als PDF herunterladen können.

4.1 Datentypen

In den vorherigen Beispielen haben wir bereits Variablen verwendet, ohne genau zu sagen, worum es sich dabei handelt. Eine Variable ist, einfach gesagt, ein Platzhalter für einen bestimmten Wert. Um zu wissen, um was für eine Art Wert es sich handelt, hat eine Variable in C immer auch einen sogenannten Typ. Wir haben auf Seite 5 schon verschiedene Datentypen vorgestellt, einige von ihnen werden auch in der Sprache C unterstützt.

Der vielleicht am häufigsten verwendete Datentyp ist `int`, der für ganze Zahlen (die wir auch *Integer* nennen) verwendet wird. Für Kommazahlen gibt es die Datentypen `float`und `double` Die beiden Typen unterscheiden sich darin, wie genau eine Zahl gespeichert werden kann. Stellen Sie sich vor, Sie möchten mit der Zahl π rechnen. Wie Sie sicher wissen, ist π eine irrationale Zahl. Sie kann nicht durch einen Bruch dargestellt werden und hat daher unendlich viele Stellen hinter dem Komma. Da Ihr Computer aber nur endlich viel Speicherplatz hat, können Sie Zahlen wie π nie ganz genau speichern. Kommt es Ihnen nicht so sehr auf Genauigkeit an, können Sie den Typ `float`verwenden. Ansonsten wählen Sie `double`

Neben den ganzen Zahlen und den Kommazahlen, gibt es noch einen weiteren Datentyp, `char`(das steht für das englische Wort *Character*), der für das Speichern einzelner Zeichen verwendet wird. Eigentlich ist aber ein `char`kein „echtes“ Zeichen, sondern einfach eine kleine Zahl im Wertebereich -128 bis 127. Wenn der Computer mit Zeichenketten arbeitet, verwendet er eine sogenannte *Kodierung*, d.h., er ordet jedem Zeichen eine eindeutige Zahl zu. Da es nur wenige mögliche Zeichen gibt, kommt der Datentyp mit wenigen Werten aus. Und das ist auch der Grund, warum man Zeichen nicht einfach auch als `int`abspeichert. Ein `char`kommt mit viermal weniger Speicherplatz als ein `int`aus. Das macht einen großen Unterschied, besonders wenn Sie es mit längeren Texten zu tun haben.

Auf Seite 5 habe ich Ihnen auch einen Datentyp vorgestellt, der Wahrheitswerte speichert, also nur die Werte *Wahr* oder *Falsch* annehmen kann. Tatsächlich gibt es einen solchen Datentyp in vielen Sprachen, z. B. auch in der Programmiersprache C++. In der Sprache C hingegen ist kein solcher Datentyp vorgesehen. Sie können aber Wahrheitswerte ganz einfach in einen ganzzahligen Datentyp, z. B. `int` oder `char` speichern. Dabei steht die 0 immer für eine logisch falsche Aussage, eine Zahl ungleich 0 bedeutet, die Aussage ist wahr.

Nicht nur Variablen haben festgelegte Typen, auch die konstanten Werte, die Sie in Ihrem Programm benutzen. Diese Konstanten heißen auch **Literale** und sie müssen so geschrieben werden, dass sich Ihr Typ sofort erkennen lässt. Am einfachsten sind die Integer-Werte, die einfach als normale Dezimalzahlen ausgeschrieben werden. Für Kommazahlen verwendet C (wie in den meisten englischsprachigen Ländern üblich) einen Dezimalpunkt und kein Dezimalkomma. Eins Komma Zwei, wirt also `1.2` geschrieben. Wir haben bereits gesehen, dass Zeichenketten in C in doppelten Anführungszeichen geschrieben werden, also z. B. `"hello, world"`. Einzelne Zeichen hingegen werden in einfachen Anführungszeichen gesetzt, also z. B. `'A'`, `'b'` oder `'!'`.

Jede Variable, die man im Programm verwenden möchte, muss zunächst eingeführt werden, man sagt auch, die Variable wird *dekla-*

riert. Bei einer **Deklaration** wird als Erstes der Typ der Variablen und dann ihr Name genannt[7]. Direkt im Anschluss kann die Variable auch direkt mit einem Wert versehen werden. Das ist dann die sogenannte **Initialisierung**.

Im folgenden Beispiel werden die `int`-Variablen `a` und `b` deklariert. `b` wird bei der Deklaration direkt initialisiert. `a` wird nach der Deklaration mittels eine Zuweisung initialisiert.

```
int a;
int b = 0;
a = 42;
```

4.2 Operatoren

Aus der Schulmathematik kennen Sie verschiedene Operatoren, einige von diesen werden auch in der Sprache C unterstützt. Dabei unterteilt man die Operatoren in mehrere Klassen. Aus der Mathematik am vertrautesten sind Ihnen vermutlich die **arithmetischen Operatoren** $+$ (*Plus*), $-$ (*Minus*), $*$ (*Mal*) und $/$ (*Geteilt*). Wenn Sie z. B. eine Variable `int zaehle`rangelegt haben, können Sie diese mit der *Anweisung* `zaehler= zaehler+ 1;` um den Wert Eins erhöhen. Eine weitere Operation, die relativ häufig eingesetzt wird, ist die *Modulo*-Operation, mit der man den Rest einer ganzzahligen Division berechnen kann. Der Modulo-Operator in C ist das $\%$-Zeichen. Wenn die Variable `x` beispielsweise den Wert 5 hat und Sie berechnen `y=x%2`, so bekommt `y` den Wert 1 zugewiesen. Auf diese Weise können Sie z. B. berechnen, ob ein Wert `x` gerade oder ungerade ist.

Neben den arithmetischen gibt es noch eine Reihe weiterer Operatoren: Den **Zuweisungsoperator** `=` haben wir schon mehrfach gesehen. Um Werte zu vergleichen, gibt es verschiedene **Vergleichsoperatoren** $>$ (*Größer*), $>=$ (*Größer oder gleich*), $<$ (*Kleiner*), $<=$ (*Kleiner oder gleich*) und $!=$ (*Ungleich*). Dazu kommt noch der Vergleichsoperator $==$ (*Gleich*), mit dem man überprüft, ob zwei Werte gleich sind. Achtung: bei diesem Operator müssen zwei Gleichheitszeichen verwendet werden, da das einzelne Gleicheiszeichen ja schon für den Zuweisungsoperator verwendet wird!

Das Resultat einer Vergleichsoperation ist immer ein Wahrheitswert, also entweder 0 für *Falsch* oder 1 für *Wahr.* In unseren Programmen werden wir Vergleichsoperatoren häufig dazu verwenden, um Bedingungen zu überprüfen. Was ist aber, wenn wir mehrere Bedingungen zu einer größeren Aussage verbinden wollen, also wenn wir z. B. erfragen wollen, ob die Werte der Variablen `x` und `y` beide positiv sind? Dazu gibt es die sogenannten logischen Operatoren `&&` (*Und*) und

[7] Wir werden später sehen, dass sich die genauen Eigenschaften von Variablen noch durch eine Reiche von Zusatzinformationen verändern lassen.

`||` (*Oder*). Im Beispiel ist der Ausdruck `x>0 && y>0` nur dann *wahr*, wenn sowohl x als auch y größer Null sind. Beachten Sie auch hier, dass die Zeichen `&` und `|` doppelt schreiben müssen. Es gibt nämlich noch die sogenannten **Bit-Operatoren** unter denen auch die Operatoren `&` und `|` (als einzelne Zeichen geschrieben) sind.

Wenn Sie an dieser Stelle etwas verwirrt sind sich fragen, warum die einzelnen Operatoren so ähnliche Zeichen verwenden, kann ich Sie gut verstehen. Für geübte Entwickler hat die knappe Schreibweise der einzelnen Operatoren den Vorteil, dass der Quellcode sehr kompakt ist. Für Programmieranfänger ist dies aber durchaus fehleranfällig. Wenn Sie sich anfangs bei der Bedeutung der Operatoren noch unsicher sind und einen falschen Operator benutzen, kann es sein, dass Sie trotzdem ein *gültiges* Programm geschrieben haben. Das ist sehr tückisch, denn Ihr Programm lässt sich damit ohne Probleme übersetzen. Der Fehler taucht vielleicht erst zur Laufzeit auf und zeigt sich in einem unerwarteten Verhalten des Programms. Für den Entwickler kann es dann sehr zeitaufwendig sein, den tatsächlichen Fehler zu finden.

4.3 Ablaufsteuerung

Wir können nun also mit Vergleichsoperatoren Bedingungen überprüfen und feststellen, ob ein bestimmter Zustand gilt oder nicht gilt. Aber wozu ist das gut? In nahezu jedem *nützlichen* Programm gibt es Teile, die nur ausgeführt werden sollen **falls** oder **solange** eine bestimme Bedingung erfüllt ist. Nehmen wir an, Sie haben ein Programm geschrieben, mit dem Sie die Zeichen in einer Textdatei zählen können. Dieses Programm bekommt als Argument einen Dateinamen mit der entsprechenden Textdatei übergeben[8]. Nun sollte Ihr Programm zuerst überprüfen, *ob* der Dateiname zu einer echten Datei gehört und *ob* Sie Berechtigung haben, diese Datei zu lesen. Wenn Sie die Zeichen zählen wollen, können Sie die Textdatei *solange* Zeichen-für-Zeichen durchgehen, bis Sie am Ende angekommen sind.

Die *ob*-Fragen programmieren Sie in C (und vielen anderen Sprachen) als sogenannte bedingte Anweisung (`if`) oder Verzweigung (`if-else`) . Mit der **if-Anweisung** überprüfen Sie eine Bedingung und führen die folgende Anweisung nur dann aus, wenn die Bedingung erfüllt ist.

Quellcode 4.1: if-Anweisung

```
1 if (x>0)
2     y = y * 2;
```

[8] Wir werden in der Vorlesung behandeln, wie man Argumente an ein Programm übergibt.

In Quellcode 4.1 sehen Sie eine `if Anweisung`Die Bedingung, die überprüft wird, lautet `x>0`. Nur in dem Fall, dass der Wert der Variablen `x` größer als 0 ist, wird die Anweisung `y = y * 2` ausgeführt.

Im Normalfall möchte man nach der Überprüfung einer Bedingung nicht nur eine einzelne, sondern gleiche eine Reihe von Anweisungen ausführen. Hierzu kann man in C Programmteile in einen Block zusammenfassen. Ein Block wird in geschweifte Klammern eingeschlossen und immer zusammenhängend ausgeführt. Blöcke können dabei wieder andere Blöcke enthalten. Damit das im Programmcode nicht zu unübersichtlich wird, rückt man Blöcke (je nach Verschachtelungstiefe) mit Leerzeichen oder Tabulatoren ein. Ohne diese Einrückungen ist der Quelltext eines normalen Programms so gut wie nicht lesbar!

Quellcode 4.2 zeigt ein Beispiel für eine Erweiterung der bedingten Anweisung, der sogenannten Verzweigung oder auch **`if`–`else`**Anweisung. Wieder wird die Bedingung `x>0` überprüft, für den Fall, dass die Bedingung wahr ist führen wir einen Block aus 2 Anweisungen aus. Die Erweiterung ist hier der sogenannte *Falsch*- oder **`else`**Teil, der nur ausgeführt wird, falls die Bedingung *nicht* erfüllt ist. In Quellcode 4.2 besteht der **`else`**Teil aus nur einer Anweisung.

Quellcode 4.2: if-else-Anweisung

```
1 if (x>0)
2 { // Mit { beginnt der Block
3     y = y  * 2;
4     z = y / 2;
5 } // Mit } endet der Block
6 else
7     z = z * 4;
```

Im einführenden Beispiel hatten wir noch den Fall erwähnt, dass ein Teil des Programms ausgeführt wird, solange eine Bedingung gilt. Bei diesem Konstrukt handelt es sich um sogenannte Programmschleifen. Wie die bedingte Ausführung sind auch Schleifen allgegenwärtig in der Programmierung und die allermeisten Programme verbringen weit mehr als 99% ihrer Laufzeit innerhalb von Schleifen.

Einen Schleifentyp, den fast alle imperativen Programmiersprachen unterstützen, ist die **`while`**Schleife. Quellcode 4.3 zeigt ein Beispiel für eine **`while`**Schleife mit der Bedingung `x>0`. Solange diese Bedingung gilt, wird der Anweisungsteil der Schleife (also wiederum eine einzelne Anweisung oder ein Block) die Schleife ausgeführt. Das bedeutet, wenn der Programmablauf am Ende des Anweisungsteils angekommen ist, springt der Programmablauf wieder zu der Bedingung. Ist die Bedingung immer noch erfüllt, so wird der Anweisungsteil erneut komplett ausgeführt.

Quellcode 4.3: while-Schleife

```
1 int x = 0;
2 while (x<10) {
3     x = x + 1;
4     printf("x ist jetzt %d\n", x);
5 }
6 y = 0;
```

Im Beispiel 4.3 ist die Bedingung anfangs erfüllt und daher wird auch der Anweisungsteil ausgeführt. x wird also um 1 erhöht und dann wir der Text *x ist jetzt 1* ausgegeben. Da x nun den Wert 1 hat, wir der Anweisungsteil erneut ausgeführt und *x ist jetzt 2* ausgegeben. Das geht solange so weiter, bis x von 9 auf 10 gesetzt wird. Es wird noch einmalig eine Ausgabe *x ist jetzt 10* ausgegeben. Zwar ist bereits nach der Erhöhung von x die Bedingung nicht mehr erfüllt, aber wir befinden uns ja aktuell noch in dem Anweisungsteil der vorherigen Schleifenüberprüfung. Erst nach dem Anweisungsteil wird die Schleife beendet und das Programm macht mit der Anweisung `y=0` weiter.

Vielleicht können Sie schon erkennen, was passieren würde, wenn wir x im Anweisungsteil selbst nicht verändern. Das Programm könnte die Schleife dann nicht mehr verlassen. In dem Fall spricht von einer *Endlosschleife*, die meistens vom Programmierer nicht erwünscht ist. Falls Ihnen einmal ein Fehler unterläuft und Ihr Programm nicht mehr selbständig zum Ende findet, können Sie mit der Tastenkombination Ctrl + C das komplette Programm (über das Betriebssystem) beenden.

4.4 Ein- und Ausgabe auf der Kommandozeile

Wir haben bereits im ersten *Hello World* Beispiel eine Ausgabe gesehen; eigentliche besteht das Programm sogar nur aus einer Ausgabe. Von einer *Ein-/Ausgabe* spricht man beim Programmieren immer dann, wenn Daten von oder auf *irgendetwas* gelesen bzw. geschrieben werden, dass nicht der Hauptspeicher (RAM) ist. Dazu zählen der Sekundärspeicher (z.B. Festplatte), das Netzwerk und die sogenannte *Standardein-/ausgabe*. Normalerweise ist die Standardausgabe das Kommandozeilenfenster, in dem Sie Ihr Programm ausführen. Manchmal übernimmt die Ausführung des Programms Ihre Programmierumgebung (also die IDE). Dann sieht es so aus, als stellt die IDE die Ausgabe dar.

Die Standardeingabe ist ebenfalls die Kommandozeilen, auf der Sie mit Ihrer Tastatur Eingaben *eintippen* können. Wenn Sie in Ihrem Programm Eingaben abfragen, müssen Sie bedenken, dass der Computer Eingaben von der Tastatur immer als *Zeichen* sieht. Also, selbst wenn Sie `123` eingeben, ist das für den Computer erst einmal keine Zahl, sondern ein Text mit drei Zeichen. Man muss in C also immer mit angeben, welche *Art* (bzw. welchen Typ) von Eingabe das Pro-

gramm erwartet. Und dabei kann eine Menge schiefgehen, vor allem, wenn sich der Benutzer nicht genau die Art von Eingabe hält, die das Programm erwartet. Gute Programme sichern daher die Benutzereingabe ab, indem Sie überprüfen, ob die tatsächliche Eingabe zu der erwarteten Eingabe passt. Ist das nicht der Fall, wird der Benutzer informiert und die Eingabe erneut abgefragt.

Im folgen Beispiel 4.4 sehen Sie, wie man in C eine Zahl vom Benutzer erfragen kann. Sie sollten immer mit einer Ausgabe starten, die klar erklärt, welche Eingabe im Folgenden erwartet wird. Das Einlesen des Werts erfolgt dann mit der Funktion `scanf`.

Quellcode 4.4: Ein-/Ausgabe

```
1 int ganzzahl = 0;
2 float kommazahl = 0;
3 printf("Gib eine ganze Zahl ein:");
4 scanf("%d", &ganzzahl);
5 printf("Gib eine irgend eine Zahl ein:");
6 scanf("%f", &kommazahl);
7 printf("Die erste Zahl ist %d, die zweite %f\n", ↙
      ganzzahl, kommazahl);
```

`scanf`funktioniert so ähnlich, wie die Ausgabefunktion **`printf`** Auch hier gibt man als erstes Argument eine Zeichenkette an. Allerdings besteht diese Zeichenkette aus Formatierungssymbolen, die bestimmen, welche Art von Eingabe gelesen werden soll. In unserem Beispiel ist es im ersten Fall eine ganze Zahl (dafür steht das Symbol `%d` wie *decimal*) und im zweiten Fall eine Kommazahl (Symbol `%f` wie *float*). Achten Sie bitte darauf, das &-Zeichen vor dem Variablennamen anzugeben. Warum das notwendig ist und was dieses & bedeutet, lernen Sie in der Vorlesung.

Wenn Sie Ihre Programme etwas *interaktiver* gestalten wollen und nicht alle Werte im Programm selbst festlegen möchten, können Sie Eingaben dieser Art verwenden. Probieren Sie auch ruhig aus, was passiert, wenn Sie *falsche* Eingaben verwenden. Sie werden später lernen, wie man diese unerwarteten Eingaben *behandelt*, indem man Sie verwirft und den Benutzer nach einer erneuten Eingabe fragt.

4.5 Felder (Arrays)

Wir haben bereits Variablen als Platzhalter für (einzelne) Werte kennengelernt. In manchen Fällen möchte man aber unter einem Namen nicht nur einen Wert, sondern eine ganze Reihe von Werten abspeichern. Nehmen wir das Spielarten-Beispiel aus Kapitel 1. Wenn wir einen Kartenstapel im Programm verarbeiten wollen, ist normalerweise entscheidend, an *welcher Stelle* ein Kartenwert steht. Die Kartenwerte und auch die Anzahl der Karten sind vorab bekannt, im

Verlauf ändert sich nur die Reihenfolge der Karten. Wenn man die Karten über Variablen abspeichern würde, wären 32, 55 oder je nach Spiel auch mehr einzelne Variablen notwendig. Viel besser wäre es, eine einzelne Variable mit k Positionen zu haben, wobei k im Beispiel die Anzahl der Karten ist.

Genau dazu gibt es in vielen Programmiersprachen die sogenannten Felder oder **Arrays**. Ein Array deklariert man wie eine Variable, gibt aber zusätzlich noch die Anzahl der Position an. Alle Einträge in einem Array besitzen den gleichen Datentyp. Ein einzelner Eintrag kann über den sogenannten *Index* erreicht werden.

Im Beispiel 4.5 deklarieren wir ein `int` Array mit 12 Positionen. An den 12 Positionen des Arrays wollen wir die Anzahl der Tage im entsprechenden Monat speichern. In C kann man alle Werte eines Arrays gleichzeitig initialisieren, wenn man die Werte als kommagetrennte Folge innerhalb von geschweiften Klammern angibt. Die direkte Initialisierung ist aber eher unüblich. Meistens initialisiert man ein Array mittels einer Schleife.

Quellcode 4.5: Arrays

```
1 int monatstage[12] = {31,28,31,30,31,30,31,31,30,31,30,31}
2 printf("Der August hat %d Tage\n", monatstage[7]);
3 monatstage[1] = 29; // Im Schaltjahr
```

Wir sehen in Zeile 2 wie lesend auf eine ein Array zugegriffen wird. Der Wert für den Monat August ist unter der Position mit dem Wert 7 zu finden (man sagt auch bei *Index 7*), weil die Stellen beginnend mit der Nummer 0 gezählt werden. Dementsprechend existiert auch der Index 12 nicht im Feld `monatstage`Die Werte eines Arrays können auch geschrieben, d. h. verändert werden. Auch hier muss die Position innerhalb des Feldes über einen Index in eckigen Klammern nach dem Array-Namen angegeben werden. In Zeile 3 wird die Anzahl der Monatstage für den Februar auf den Wert 29 gesetzt, was im Fall eines Schaltjahres auch zutrifft.

4.6 Weitere Themen

An dieser Stelle endet unser kurzer Überblick über die Sprache C. Sie haben nun das Rüstzeug, um kleinere Programme selbstständig zu schreiben. Um Software für „echte“ Anwendungsfälle entwickeln zu können, sollten Sie aber unbedingt weitere Sprachkonstrukte von C und dann auch von C++ kennenlernen.

In den kommenden Vorlesungen *Programmierung mit C++ 1 und 2* werden Sie erfahren, wie diese Konstrukte aussehen und wie man Sie anwendet. Sie lernen, wie man **Zeichenketten** verarbeitet und Daten mittels **Dateien** auf der Festplatte liest und schreibt. Außerdem

werden wir einem Datentyp behandeln, für den die Programmiersprache C bekannt und teilweise auch gefürchtet ist: die sogenannten Zeiger (engl. **Pointer**). Zeiger benötigt man vor allem, um mit der sogenannten **dynamischen Speicherverwaltung** umzugehen. Außerdem werden wir in der Vorlesung betrachten, wie man bestimmte **Datenstrukturen** und **Algorithmen** in der Sprache C programmiert.

4.7 Lernkontrollfragen und Aufgaben

a. Schreiben Sie ein Programm, mit dem Sie die Summe der Zahlen von 1 bis 100 berechnen können.
b. Was ist das Problem bei folgendem Programmcode:

```
int x = 0;
int y = 0;
while (x<10) {
        y = y + 1;
}
```

c. Deklarieren Sie ein Array mit 100 `int` Elementen. Initialisieren die Werte des Arrays mit dem konstanten Wert 1.
d. Deklarieren Sie ein Array mit 100 `int` Elementen. Initialisieren die Werte des Arrays mit den Zahlen 1 bis 100.
e. In C gibt es keinen eingebauten Operator für die Potenzfunktion $f(x) = x^n$ (mit $n \in \mathbb{N}$). Wie könnten Sie eine Potenz (z. B. 2^8) mit den Ihnen bekannten Konstrukten ausrechnen?

Wenn Ihnen die Aufgaben bis hierhin wenig Probleme bereitet haben, setzen Sie Ihre Arbeit mit den folgenden (etwas schwierigeren) Aufgaben fort.

f. In Ihrem Programm gibt es eine `double`Variable `my_double`mit dem Wert 1.9 und eine `int` Variable `my_int`mit dem Wert 0. Nun machen Sie eine Zuweisung `my_int = my_double`Welchen Wert könnte `my_int`danach haben?
g. Schreiben Sie ein Programm, mit dem Sie Ihre Vermutung für f. überprüfen können.
h. Erweitern Sie das Programm aus Aufgabe d. so, dass im Array an der Position x nicht der Wert $x + 1$ steht, sondern die Summe aller Werte von 0 bis $x + 1$. Also 1, 3, 6, 10, 15, ...
i. Erweitern Sie das Programm aus Aufgabe e. so, dass nach der Berechnung der Potenz überprüft wird, ob das Ergebnis stimmt. Rechnen Sie das gewünschte Ergebnis dazu vorher „per Hand“ aus und vergleichen Sie dies mit dem vom Programm errechneten Wert. Geben Sie die Meldung `"Das Ergebniss stimmt"` aus, wenn Ihr Prgramm richtig rechnet. Ansonsten geben Sie eine Fehlermeldung aus.

j. Wie können Sie die *Modulo*-Operation verwenden, um zu überprüfen, ob der Wert einer `int` Variablen ungerade ist?

5 Eine erste Anwendung programmieren

Bei der Bearbeitung der vorherigen Kapitel haben Sie ein „Hello World“-Programm und vielleicht noch weitere kleinere Programme geschrieben, übersetzt und getestet. In diesem Kapitel wollen wir eine etwas größere Anwendung vorstellen, die Sie im Rahmen des Workshopes (und auch gerne danach) implementieren können. Wenn Sie die unten stehenden Aufgaben nicht oder noch nicht selbstständig bearbeiten können, ist das kein Problem.

Sie werden im Verlauf der Vorlesung die notwendigen Fertigkeiten erlangen um mindestens die ersten Aufgaben eigenständig lösen zu können. Es ist aber in jedem Fall hilfreich, wenn Sie schon jetzt mit der Aufgabenstellung und den vorgegeben Beispielen experimentieren. Nutzen See die Gelegenheit, Ihre Tutoren oder Ihre Kommilitonen zu fragen und arbeiten Sie gerne in Teams an einer gemeinsamen Lösung.

5.1 Tic-Tac-Toe

Sie kennen sicher das Spiel *Tic-Tac-Toe* (auch *3-gewinnt* genannt), bei dem 2 Spieler auf einem Spielfeld mit 3×3 Feldern versuchen müssen, drei Felder in einer Reihe zu belegen. Tic-Tac-Toe ist vielleicht nicht das spannendste Spiel – bei geübten Spieler kommt es eigentlich immer zu einem unentschieden – dafür ist es aber recht einfach und daher mit überschaubarem Aufwand zu programmieren.

Zunächst wollen wir überlegen, wie wir ein Spielbrett mit unseren bisherigen Kenntnissen „zeichnen“ können. Für ein grafische oder sogar eine 3D Oberfläche reichen unsere aktuellen Mittel noch nicht aus. Aber wir haben ja die `printf`Anweisung, mit der wir auf die Eingabeaufforderung drucken können. Unter Hinzunahme einiger Sonderzeichen lässt sich auch damit ein Spielbrett zeichnen, das zumindest seine Funktion erfüllt.

Quellcode 5.1 zeigt die Implementation der Funktion `zeichne_spielfeld`, mit der man die Bildschirmausgabe für das Tic-Tac-Toe Programm erzeugen kann. Die Funktion besitzt 2 Parameter, den aktuellen Zustand des Spielfelds sowie die aktuelle Nummer des Zugs.

Als Erstes stellt sich dabei die Frage, wie man den Zustand des Spielfelds abspeichern kann. Wenn wir die Felder des Spielplans durchnummerieren, können wir das als Array mit 9 Einträgen abbilden. Bleibt noch die Frage, welchen Datentyp wir für die einzelnen Spielfelder wählen. Hier wäre z. B. `int` denkbar. Die Kodierung könnte dann lauten $0 \Rightarrow$ leeres Feld, $1 \Rightarrow$ X und $2 \Rightarrow$ O.

Die Funktion `zeichne_spielfeld` erwartet allerdings eine andere Kodierung des Spielplans. Wir legen das Spielfeld als Array von Zeichen (also `char`) an und belegen jedes Feld mit dem Zeichen, dass auf dem Feld erscheinen soll. Das sind am Anfang, vor dem ersten Zug, die Ziffern 1 bis 9. Damit weiß der Benutzer des Programms sofort, welche Nummer ein entsprechendes Feld hat. Platziert ein Spieler sein Symbol (*X* oder *O*) auf einem Feld, so soll das Symbol bei der nächsten Ausgabe des Spielbretts auf dem Feld erscheinen.

In dem Code-Beispiel 5.1 sind noch einige weitere Details, auf die wir hier nicht näher eingehen. Zu nennen ist etwa das Löschen der Bildschirmausgabe mit dem Systemaufruf `system` sowie das Abwechseln zwischen den Spielern *X* und *O* mit der Modulo-Funktion. Sie können gerne mit dem Beispiel experimentieren und versuchen den Code zu ändern oder zu erweitern. Rufen Sie die Funktion in einer $main$-Funktion auf und beobachten Sie, welche Veränderungen sich im Verhalten zeigen.

Quellcode 5.1: Funktion zum Zeichnen des Tic-Tac-Toe Spielfelds

```
1  void zeichne_spielfeld(char spielfeld[], int zug) {
2      system("clear");
3      printf("  Tic Tac Toe\n\n");
4      printf("  SpielerIn 1 (X)  gegen  SpielerIn 2 (O)\n");
5      if(zug==0) {
6          printf("  SpielerIn 1 (X) beginnt\n\n");
7      }
8      else {
9          if(zug%2==0)
10             printf("  SpielerIn X am Zug\n\n");
11         else
12             printf("  SpielerIn O am Zug\n\n");
13     }
14
15     printf("       |     |     \n");
16     printf("    %c  |  %c  |  %c \n", spielfeld[0], ↙
           spielfeld[1], spielfeld[2]);
17     printf("  _____|_____|_____\n");
18     printf("       |     |     \n");
19     printf("    %c  |  %c  |  %c \n", spielfeld[3], ↙
           spielfeld[4], spielfeld[5]);
20     printf("  _____|_____|_____\n");
21     printf("       |     |     \n");
22     printf("    %c  |  %c  |  %c \n", spielfeld[6], ↙
           spielfeld[7], spielfeld[8]);
23     printf("       |     |     \n\n");
24 }
```

5.2 Eingabe von Zeichen

Bevor wir uns überlegen können, wie wir das Tic-Tac-Toe Spiel implementieren, müssen wir zunächst eine wichtige Funktion kennenlernen, die für die Interaktion mit Programmen sehr wichtig ist. Bisher können wir nur Daten auf dem Bildschirm ausgeben; für ein Spiel muss der Benutzer aber auch in der Lage sein, **Eingaben** in das Programm zu tätigen. Bei Programmen die in der Eingabeaufforderung laufen, kommen die Eingaben i. d. R. von der Tastatur. Wie bei `printf`, benutzt man zur Tastatureingabe Funktionen, die zumeist aus der Bibliothek `stdio` stammen.

Wenn Sie noch ungeübt sind und mit Tastatureingaben in C experimentieren, werden Sie vielleicht häufiger ein unerwartet Verhalten Ihres Programms erleben. Z. B. Lesen Sie an einer Stelle in Ihrem Programm eine Tastatureingabe ein, die Sie bei der vorherigen Eingabe schon eingetippt haben. Das hat damit zu tun, wie ihr Programm vom Betriebssystem die Zeichen von der Tastatur abfragt. Daher ist zu Anfang am besten, wenn Sie für die Tastatureingabe Beispielcode verwenden, der robust funktioniert. In Quellcode 5.2 ist eine Funktion `lese_ein_zeichen` abgebildet, mit der ein einzelnes Zeichen von der Tastatur gelesen werden kann. Das Beispiel benutzt die Funktion `getchar()` mit der ein Zeichen aus der Eingabe gelesen werden kann.

Wenn die Eingabe noch leer ist, Sie also noch nichts eingetippt haben, wartet `getchar()` auf die betätigung der Enter-Taste. Nun kann es aber sein, dass Sie mehrere Zeichen eingetippt haben. Jeder Aufruf von `getchar()` liefert Ihnen nun nach und nach alle Zeichen. Aber vermutlich wollten Sie nur, dass das erste Zeichen gelesen wird. Aus diesem Grund verwirft die Funktion `lese_ein_zeichen` alle Zeichen, die nach dem ersten (bis einschließlich dem Enter-Zeichen) folgten.

Quellcode 5.2: Funktion zum Einlesen eines Zeichens von der Tastatur

```
1  char lese_ein_zeichen() {
2      // Erstes Zeichen einlesen:
3      char resultat = getchar();
4      // Alle weiteren Zeichen bis zum Zeilenende verwerfen:
5      while ( getchar() != '\n' );
6      // Erstes Zeichen zurueckgeben:
7      return resultat;
8  }
```

Aufgabe 1: Schreiben Sie ein Programm, dass Tastatureingaben (ein einzelnes Zeichen) erfragt und direkt ausgibt. Schauen Sie sich im Quellcode 5.1 ab, wie man mit `printf` einzelne Zeichen formatiert ausgeben kann. Achten Sie dabei auf das Formatierungszeichen `%c`. Wenn Ihr Programm für eine Eingabe funktioniert, erweitern Sie es so, dass Sie 10 Eingaben in Folge (jeweils mit einer Ausgabe für jede

Eingabe) abfragen können. Die Ausgabe soll wie im folgenden Beispiel aussehen:

```
Eingabe:a
Das Zeichen ist: a
Eingabe:B
Das Zeichen ist: B
Eingabe:CDE
Das Zeichen ist: C
Eingabe:x
Das Zeichen ist: x
Eingabe:235
Das Zeichen ist: 2
...
```

5.3 Eingabe der Spielzüge

Sobald wir einzelne Zeichen von der Tastatur einlesen können, kann man eine Computer-Variante des Brettspiels programmieren. Die größte Einschränkung momentan ist, dass das Programm noch zwei menschliche Spieler erwartet und nicht selbständig spielen kann. Nach dem ersten Semester, sollten Sie in der Lage sein, auch ein Tic-Tac-Toe Programm zu schreiben, das eigene Züge durchführen und so gegen einen Menschen antreten kann.

Aufgabe 2: Schreiben Sie ein Programm, dass den Spielplan des Tic-Tac-Toe-Feldes ausgibt und den aktuellen Spieler nach einer Eingabe fragt. Sobald der Spieler seine Eingabe getätigt hat, soll der aktualisierte Spielplan ausgegeben werden. Danach ist der andere Spieler am Zug.

5.4 Fehlerhafte Eingaben behandeln

Immer, wenn Sie Benutzereingaben in Ihrem Programm verwenden, müssen Sie auch mit fehlerhaften Eingaben rechnen. Sie erwarten z. B., dass der Spieler eine Feldnummer im Bereich 1 bis 9 eintippt, bekommen dann aber von der Benutzer die Eingabe 0. Oder der Spieler möchte das *X* (oder das *O*) auf einem bereits belegten Feld platzieren.

Ihr Programm sollte mit solchen fehlerhaften Eingaben umgehen können, den Benutzern eine Fehlermeldung ausgeben und bestenfalls zu einer erneuten Eingabe auffordern. Ganz generell ist es eine wichtige Aufgabe der Softwareentwickler, mögliche Problemzustände im Programm zu erkennen bzw. auszuschließen.

Aufgabe 3: Erweitern Sie das Tic-Tac-Toe Programm so, dass möglichst viele fehlerhafte Eingaben erkannt und behandelt werden.

5.5 Gewinnsituation erkennen

Bisher ist das Programm eine Computer-Version eines Tic-Tac-Toe Spielbretts. Die Spieler können Züge eingeben und bekommen das aktuelle Spielbrett angezeigt. Eine erste Erweiterung, die ein “echtes„ Spielbrett nicht liefern kann, ist zu erkennen, ob ein Spieler gewonnen hat.

Ab dem fünften Zug kann es theoretisch zu einer Gewinnsituation kommen, wenn ein Spieler 3 Zeichen gesetzt hat. Sind diese Zeichen in der gleichen Zeile, der gleichen Spalte oder in einer Diagonalen angeordnet, so ist das Spiel vorbei.

Aufgabe 4: Erweitern Sie das Tic-Tac-Toe Programm so, dass das Programm nach jedem Zug überprüft, ob eine Gewinnsituation vorliegt. Das Programm soll in dem Fall den Gewinner ermitteln und beglückwünschen. Danach soll das Programm beendet werden.

5.6 Erweiterungen

Wenn das Tic-Tac-Toe Programm für zwei Spieler vollständig funktioniert, ist der nächste Schritt, das Programm selbständig spielen zu lassen. Diese Erweiterung zu implementieren übersteigt den Rahmen dieses Workshopes. Allerdings ist es interessant sich zu überlegen, *wie* ein Computer erfolgreich gegen einen Menschen spielen kann. Also wie soll der Computer, basierend auf einem aktuellen Spielzustand, einen guten Zug berechnen? Das Problem, gute Entscheidungen zu Treffen, stellt sich nicht nur bei Spielen, sondern bei sehr vielen Anwendungen der Informatik und viele theoretischen Arbeiten und Methoden beschäftigen sich mit dieser Fragestellung.

Für das Tic-Tac-Toe Problem könnte man z. B. folgende Ansätze verfolgen:

1. Wähle einen *zufälligen, gültigen Zug* aus.
2. Wähle den besten Zug nach fest einprogrammierten Regeln aus.
3. Simuliere den Spielverlauf über die nächsten möglichen Züge und wähle den Zug aus, der zu der besten Stellung in der Zukunft führt.
4. Zeichne sehr viele Tic-Tac-Toe Spiele auf und notiere das Resultat. Schreibe ein Programm, dass aus diesen *Daten* eine gute Spielstrategie ermittelt und nutze diese Strategie in dem Tic-Tac-Toe Programm.
5. Lasse 2 Computerspieler gegeneinander antreten, die nach einer der vorherigen Strategien spielen. Benutze die Spielergebnisse, um die Spielstrategie zu verbessern.

Der erste Ansatz ist noch mit sehr geringem Aufwand zu implementieren. Lange Zeit waren Strategien, die nach einprogrammierten Regeln oder mit der Simulation von Folgezügen arbeiten der Stand der Technik. Ein Nachteil dieser Ansätze ist es, dass Programmierer die Anwendung sehr gut verstehen müssen, was bei Tic-Tac-To noch recht einfach ist, bei anderen Anwendungen aber sehr komplex werden kann. Die Ansätze 4 und 5 kann man dem Gebiet der Künstlichen Intelligenz zuordnen. Statt zu programmieren, *wie* ein guter Zug berechnet wird, soll der Computer aus bestehenden Daten eine gute Handlungsstrategie *lernen*[9]. Ein spannender Ansatz ist es, die Vorgehensweisen 1 und 5 so zu kombinieren, dass ein anfangs „dummes", also zufällig spielendes Programm sich nach und nach durch das automatische Spielen vieler Partien zu verbessert.

Vielleicht greifen Sie ja einen Ansatz heraus und entwickel ein Tic-Tac-Toe Programm das gegen menschliche Spieler gewinnen kann. Ich würde mich freuen, von Ihren Ergebnissen zu erfahren und wünsche Ihnen viel Spaß beim Programmieren!

Literaturverzeichnis

[DBSG11] Manfred Dausmann, Ulrich Bröckl, Dominik Schoop und Joachim Goll. C als erste Programmiersprache. Vieweg+Teubner, 2011.

[KR88] Brian W. Kernighan und Dennis M. Ritchie. The C Programming Language (2nd ed.). Prentice Hall, 1988.

[KR90] Brian W. Kernighan und Dennis M. Ritchie. Programmieren in C (2. Auflage). Carl Hanser, 1990.

[Str13] Bjarne Stroustrup. The C++ Programming Language (4th ed.). Addison Wesley, 2013.

[Str14] Bjarne Stroustrup. Programming: Principles and Practice Using C++ (2nd ed.). Addison Wesley, 2014.

[TIO20] TIOBE Software BV. TIOBE Programming Community Index . https://www.tiobe.com/tiobe-index/, 2020. [Online; abgerufen am 07.10.2020].

[9] Der sehr bekannte Informatiker Donald E. Knuth erzählt in einem Youtube Video eine interessante Anekdote, wie er als junger Student ein Tic-Tac-Toe Programm entwickelt hat. Darin sind auch die oben genannten Lösungsansätze skizziert. Der Link zum Video lautet https://www.youtube.com/watch?v=_c3dKYrjj2Q (abgerufen am 10.09.2019).