Einführung in die maschinennahe, imperative, funktionale, relationale und objektorientierte Programmierung

-

EMIFROP 0.XX

Markus Alpers
B.Sc. und Ausbilder f. Industriekaufleute

3. März 2016

Inhaltsverzeichnis

Hinweis bezüglich diskriminierender Formulierungen

In diesem Text wurde darauf geachtet Formulierungen zu vermeiden, die diskriminierend verstanden werden können. Im Sinne der Lesbarkeit wurden dabei Formulierungen wie „Informatiker und Informatikerinnen“ durch „InformatikerInnen“ (mit großem i) ersetzt. An anderen Stellen habe ich Formen wie eine/einer durch eineR zusammengefasst. Hier berufe ich mich auf den Artikel „Sprache und Ungleichheit“ der Bundeszentrale für politische Bildung, kurz BpB, vom 16. April 2014, insbesondere auf den Absatz „Zum Umgang mit diskriminierender Sprache“, online abrufbar unter:

`http://www.bpb.de/apuz/130411/sprache-und-ungleichheit?p=all`

Sollten Sie dennoch Formulierungen entdecken, die diesem Anspruch nicht entsprechen, möchte ich Sie bitten, mir eine entsprechende Nachricht zu senden, denn es ist mir wichtig, Ihnen mit diesem Buch eine wertvolle Unterstützung beim Start in die faszinierende Welt der Informatik zu bieten. Das sollte nicht durch verletzte Gefühle in Folge missverständlicher Formulierungen torpediert werden.

Sie erreichen mich unter `markus.alpers@haw-hamburg.de`.

Hinweis zur Lizenz

zenz"´. Wenn Sie einen Teil finden, in dem diese „Hinweise zur Lizenz"´ nicht zu finden ist, dann dürfen Sie für den persönlichen Gebrauch dennoch Kopien davon anfertigen und Sie dürfen diese Kopien außerhalb von kommerziellen Projekten frei verwenden.

Hinweis zur Verwendbarkeit in wissenschaftlichen Arbeiten

Bitte beachten Sie dabei aber, dass die Verwendung dieses Textes im Rahmen wissenschaftlicher Publikationen zurzeit aus anderen Gründen problematisch ist: Wie viele andere Quellen, die frei im Internet verfügbar sind, wurde auch dieser Text bislang nicht durch einen nachweislich entsprechend qualifizierten Lektor verifiziert. Damit genügen Zitate aus diesem Band streng genommen noch nicht den Ansprüchen wissenschaftlicher Arbeiten.

Bitte beachten Sie außerdem, dass dieses Buch eine Konvention nutzt, die in wissenschaftlichen Arbeiten verpönt ist: Wenn in einer wissenschaftlichen Arbeit ein Begriff hervorgehoben wird, dann wird dazu kursive Schrift verwendet. In diesem Buch verwende ich dagegen Fettdruck, da es vielen Menschen schwer fällt, einen kursiv gedruckten Begriff schnell zu finden und ich mir wünsche, dass Sie es möglichst effizient auch als Nachschlagewerk nutzen können.

Teil I

Programmierung für Einsteiger

Kapitel 1

Typische Irrtümer darüber, was Programmieren ist.

Kapitel 2

Von der Nachrichtentechnik zur Programmierung

Kapitel 3

Vorbereitung fürs Programmieren

Kapitel 4

Ausgewählte Programmiersprachen

Kapitel 5

Grundlagen der Entwicklung verteilter Anwendungen

Kapitel 6

HTML5

Kapitel 7

CSS3

Kapitel 8

PHP 5.6

Kapitel 9

MySQL

Teil II

Fortsetzung für Studierende der Informatik

Kapitel 10

Grundlagen der imperativen Programmierung

Die Programmiermethoden, die Sie bis jetzt kennen gelernt haben entsprechend weitgehend dem, was Informatikstudierende bereits vor dem Studium beherrschen sollten. Denn genau wie ein Studium der Anglistik nicht dazu da ist, um Englisch zu lernen, sondern um sich intensiv mit den Feinheiten der englischen Sprache zu beschäftigen, geht es im Studium der (Medien-)Informatik nicht darum, das Programmieren zu lernen. Vielmehr geht es hier darum, zu lernen, welche fortgeschrittenen Konzepte es gibt, um Rechnersysteme möglichst elegant und effizient dazu zu bringen, komplexe Probleme zu lösen. In der Medieninformatik betrachten wir dabei vorrangig Probleme, die bei der Verarbeitung von audio-visuellen Daten auftreten.

Datenverarbeitung schließt die folgenden Dinge ein:

- **Erzeugung neuer Daten aus vorhandenen Daten**, ist die offensichtlichste Beschäftigung von (Medien-)InformatikerInnen.
- **Speicherung**, also die Datenübertragung über einen Zeitraum und der daraus resultierende Zeitaufwand muss dabei immer beachtet werden.
- **Datenübertragung**, also die Datenübertragung von einem Ort zum anderen führt ebenfalls zu einem Zeitaufwand, der für Software relevant ist.
  Beachten Sie dabei, dass bei jeder Operation, Funktion usw. Datenmengen vom Speicher in den Prozessor, innerhalb des Prozessors und vom Prozessor zum Speicher übertragen werden, was wir bislang bei der Programmierung vollständig ignoriert haben.

Wichtig:

Auch die Anzeige eines einzelnen Zeichens auf dem Bildschirm setzt voraus, dass wir als ProgrammiererInnen dieses Zeichen in einer vereinbarten Art und Weise (also nach einer für den Einzelfall festgelegten Codierung) vom Prozessor an den Grafikprozessor übertragen. Wenn Sie also bislang dachten, dass eine Ausgabe quasi automatisch auf dem Bildschirm angezeigt wird, dann haben Sie wie die meisten ProgrammiererInnen falsch gedacht. Wir werden uns in C bei den sogenannten Header-Dateien und in Java bei der sogenannten Klassenbibliothek einmal ansehen, wie die Laufzeitumgebung bzw. „die Programmiersprache“ uns diese Arbeit abnimmt.

Kontrolle

- Sie finden diese drei Punkte langweilig?
  Dann finden Sie die Medieninformatik langweilig und sollten sich umgehend einen anderen Studienplatz suchen, denn im Kern läuft alles was Sie als MedieninformatikerIn tun werden auf die Beschäftigung mit diesen drei Aspekten hinaus.

Denn die Methodik in der Medieninformatik ist weitgehend die gleiche wie in der Informatik. Und beim Programmieren gibt es gar keine Unterschiede. Da Sie durch den ersten Teil des Buches und die Übung zu Hause jetzt ein wenig Verständnis fürs Programmieren aufgebaut haben, könnten Sie auf die Idee kommen, dass Sie eigentlich nur mehr Übung und Kenntnisse einzelner Programmiersprachen brauchen, um ein umfassendes Verständnis von Programmierung zu erlangen. Und genau das ist falsch. Alles, was Sie damit erreichen ist die Fähigkeit, ein bestimmte Gruppe von imperativen Programmiersprachen zu beherrschen. Damit sind Sie aber bei vielen Problemen nicht im Stande, eine Programmiersprache auszuwählen, mit der Sie das Problem effizient lösen können. Streckenweise können Sie es damit überhaupt nicht lösen.

Damit Sie ein umfassendes Verständnis von Programmierung erhalten, müssen wir zunächst an den Anfang zurückkehren und einige Begriffe definieren, mit denen wir dann die Eigenschaften und Möglichkeiten verschiedenster zustandsbasierter Programmiersprachen beschreiben können. Denn nur so können Sie ohne allzu große Anstrengungen von einer imperativ streng typisierten Sprache wie C zu einer logischen Programmiersprache wie PROLOG und dann zu einer dynamisch typisierten funktionalen Programmiersprache wie JavaScript wechseln.

Das bringt uns gleich zu einer essentiellen Unterscheidung von Programmiersprachen:

- **Zustandsbasierte Programmierung** ist die Programmierung von Computern, die wie Windows-, MacOS- und Linux-Rechner zu jedem Zeitpunkt einen festen Zustand kennen. In der Medieninformatik kommen wir meist nur mit dieser Art von Rechnern in Kontakt, allerdings liegt das vorrangig daran, dass die Entwicklung nicht-zustandsbasierte Systeme ein fundiertes Verständnis der Elektro- und Nachrichtentechnik benötigen.
- **Nicht-zustandsbasierte Programmierung** ist die Programmierung von Computern, die z.B. als Steuer- und Regelsysteme bezeichnet werden. Hier erfolgt die Programmierung z.B. in Form von Gleichungen höheren Grades, die kontinuierlich ausgewertet werden. Diese Systeme kommen z.B. in Autos zum Einsatz, wo sie dann steuern, ob der Airbag ausgelöst wird. Ein anderer Einsatzbereich wäre die Steuerung des Kühlwasserkreislaufs in einem Kraftwerk.

Es ist aber wichtig an dieser Stelle zu betonen: Als angehende (Medien-)InformatikerInnen müssen Sie mehr beherrschen als nur die Programmierung in C-artigen Sprachen. Als reine/r ProgrammiererIn z.B. für Computerspiele ist es in aller Regel überflüssig. Auch für MT-Studierende, die z.B. die Programmierung in C++ für die Entwicklung von Musiksoftware erlernen wollen führt der Inhalt dieses Teils des Buches weit über das nötige Maß hinaus. Für die ist dagegen der dritte Teil dieses Buches sehr interessant: Darin geht es gerade um die Aspekte der technischen Informatik, die wir bei der Programmierung in Assemblersprachen sowie C und C++ beherrschen müssen. Dagegen ist die Entwicklung von Software für kontinuierliche (also nicht-zustandsbasierte) Systeme nicht Teil dieses Buches.

10.1 Virtuelle Objekte

Alle höheren Programmiersprachen (das sind alle Sprachen, die nicht maschinennah sind) verwenden virtuelle Objekte. Wichtig: Diese haben mit den Objekten einer objektorientieren Sprache nichts zu tun. Um den Unterschied zu verdeutlichen ist hier deshalb durchgehend die Rede von „virtuellen Objekten“. Eine Art von virtuellen Objekten sind die sogenannten **Variablen**. Virtuellen Objekte haben bestimmte Eingenschaften, die in den unterschiedlichen Programmierparadigmen leider nicht gleich bezeichnet werden. Auch um Ihnen dabei zu helfen, diese Klippe zu umschiffen, führe ich die folgende einheitliche Nomenclatur ein.

Es ist wichtig, dass Sie verstehen, dass ein virtuelles Objekte eine beliebige Kombination der folgenden Eigenschaften haben kann. Variablen sind eine Art von virtuellen Objekten, aber eben nur eine Art, die besonders bekannt ist. Andere Arten von Programmiersprachen haben andere Arten von virtuellen Objekten, die sich zum Teil deutlich von Variablen unterscheiden. So lange Sie beim Programmieren an Variablen denken und nicht verstehen, dass das nur ein Spezialfall ist werden Sie eine Vielzahl an Programmiersprachen nicht erlernen können.

10.1.1 Bezeichner

Virtuelle Objekte können einen Bezeichner haben. Das ist so etwas wie ein Name. Hier verwende ich aber bewusst den Begriff des Bezeichners, um damit von Namen und Bezeichnungen zu unterscheiden, wie wir sie im alltäglichen Sprachgebrauch nutzen.

Erinnern Sie sich bitte wieder an die anonymen Variablen: Das waren Variablen, die als Ergebnis einer Operation oder Funktion entstanden aber noch keinem Bezeichner zugeordnet wurden.

10.1.2 Pointer

Virtuelle Objekte können sich auf eine Speicheradresse des Computers beziehen. Diese Referenz auf eine Speicheradresse wird meist als Pointer bezeichnet.

In Sprachen wie Java und PHP könen wir auf den Pointer nicht zugreifen, dort existieren Pointer also scheinbar nicht. In maschinennahen Sprachen wie C und C++ ist die Beherrschung von Pointern eine essentielle Fähigkeit, die ProgrammiererInnen beherrschen müssen.

10.1.3 Wert

Virtuelle Objekte, die sich auf eine Speicheradresse beziehen können einen Wert haben, der sich aus dem Binärwert ergibt, der unter dem Pointer gespeichert ist. Das ist aber nur ein Spezialfall, nicht die Regel.

In der logischen Programmiersprache PROLOG definieren wir beispielsweise virtuelle Objekte nur anhand eines Bezeichners, der gewissermaßen immer den Wert wahr hat. Da es in dieser Sprache den Wert falsch aber gar nicht gibt, haben virtuelle Objekte dort im Grunde niemals einen Wert.

Änderbarkeit eines Wertes

Hierbei handelt es sich nicht um eine zentrale Eigenschaft, die ein virtuelles Objekt haben kann, aber da aus Komfortgründen viele Programmiersprache die Möglichkeit bieten, einen Wert als konstant festzulegen, habe ich diesen Punkt in die Aufstellung aufgenommen.

Tatsächlich ist eine Möglichkeit für Angriffe auf IT-Systeme die Tatsache, dass der Wert eines Objekts niemals wirklich statisch sein kann: Wenn ein virtuelles Objekt einen Wert hat, dann ist das (s.o.) ein Binärwert, der in einem Speicherbereich abgelegt ist. Dieser Speicherbereich selbst ist aber immer änderbar. Also gibt es Möglichkeiten, diesen Wert durch Manipulation des Rechners oder den Missbrauch der Programmiersprache zu ändern. Die Frage ist nur, wie leicht ein solcher Angriff durchführbar ist.

10.1.4 Interpretation des Wertes

Wenn ein virtuelles Objekt einen Pointer hat, dann gibt es eine Interpretation des dort gespeicherten Wertes.

Das haben Sie in PHP als Datentyp einer Variablen kennen gelernt.

10.1.5 Operation(en)

Alles, was ohne weitere Programmierung mit einem Wert getan werden kann wird als Operation bezeichnet.

Wenn Sie das missverstehen, werden Sie vielleicht auf die Idee kommen, dass Operationen und Funktionen das gleiche sind, aber das ist falsch: Operationen und Funktionen sind etwas unterschiedliches.

Wenn Sie sich jetzt fragen, was denn der Unterschied zwischen einer **Operation** und einer **Funktion** ist, folgt hier eine kurze Erläuterung:

- Eine Operation ist etwas, das Sie mit einem Wert tun können ohne das dazu irgend eine Erweiterung im Programm erstellt werden muss. Wenn Sie beispielsweise ein virtuelles Objekt haben, das als Wert eine ganze Zahl repräsentiert, dann können Sie im Regelfall die Grundrechenarten darauf ausführen. Dagegen ist das Ziehen der Wurzel etwas, bei dem mehrfach verschiedene Operationen auf einen Wert angewendet werden.
- Nehmen Sie als Beispiel das Heron-Verfahren:
  $x_{n+1} = \frac{1}{2}(x_n + \frac{x_n}{a})$, $a \in \mathbb{N}$.
  Bei jeder Iteration (wiederholte Anwendung des Verfahrens mit dem

> Ergebnis der vorigen Berechnung) wenden wir die Operationen Multiplikation, Division und Addition an. Die beschriebene Kombination dieser Operationen ist eine Funktion.

Bitte beachten Sie, dass diese Definition nicht allgemeingültig ist; es gibt noch andere Möglichkeiten, eine Funktion zu definieren. Für eine vollständige Aufstellung wenden Sie sich bitte an den Mathematik-Dozenten Ihres Vertrauens. Später werden Sie noch mehr über Funktionen als Teil von Computerprogrammen erfahren, diese Beschreibung ist also selbst im Bereich der Programmierung nicht vollständig.

Kontrolle:

Ist Ihnen klar, warum die Änderbarkeit eines Wertes keine Operation ist, die Änderung des Wertes eines Objekts dagegen schon?

10.1.6 Relation

Zwischen zwei virtuellen Objekten können feste Zusammenhänge existieren, für die es unwichtig ist, ob das virtuelle Objekt einen Wert hat. Diese Zusammenhänge werden als Relationen bezeichnet.

Wichtig: Es gibt relationale Operationen, bei denen der Wert zweier virtueller Objekte verglichen wird. Diese werden Sie als boolesche Operationen kennen lernen. Der hier eingeführte Begriff der Relation bezieht sich dagegen auf eine Relation die zwischen zwei virtuellen Objekten selbst existiert.

Auch hier gilt wieder, dass diese Definition des Begriffs einer Relation nicht vollständig ist; in der Mathematik werden Sie z.B. Relationen als eine mögliche Definition für Funktionen kennen lernen.

10.1.7 Lebensdauer oder Gültigkeitsbereich

Virtuelle Objekte werden erzeugt. Sie sind jeweils für eine bestimmte Dauer oder innerhalb eines festen Bereichs eines Computerprogramms gültig. Hierfür wird auch der Begriff des **Scope** genutzt.

Bei den Programmiersprachen, mit denen wir uns beschäftigen brauchen Sie nicht zu verstehen, wie dieser Mechanismus funktioniert, weil Sie hier nicht direkt auf Speicherzellen zugreifen. Der Mechanismus, der ein virtuelles Objekt „löscht“, der also das Ende der Lebensdauer sicher stellt, wird als **garbage collector** bezeichnet. Sobald Sie dagegen maschinennah programmieren, müssen Sie sich damit auseinandersetzen. Als Ausblick sei schon einmal gesagt, dass Sie Speicher nicht löschen, sondern nur überschreiben

können. Sie können natürlich mit einem Schweißbrenner die Speicherbausteine zerstören und damit den Inhalt wirklich löschen, aber im Alltag erscheint das doch nicht recht praktikabel. Tatsächlich basieren auf dieser Tatsache auch einige Angriffsmethoden im Bereich der IT-Sicherheit. So lässt sich beispielsweise Kältespray einsetzen, um Speicher zu fixieren, selbst wenn der Strom anschließend abgeschaltet wird. Das ist eine Möglichkeit um unter bestimmten Bedingungen ein Passwort aus einem heruntergefahrenen Rechner mit verschlüsselter Festplatte auszulesen.

10.1.8 Zugriffsrecht

Virtuelle Objekte können verwendet werden. Ob sie von jedem anderen virtuellen Objekt bzw. von jedem beliebigen Teil eines Programms verwendet werden können wird über die Zugriffsrechte festgelegt.

Neben dem Zugriffsrecht auf virtuelle Objekte gibt es noch Zugriffsrechte, die durch die Konfiguration des Betriebssystems festgelegt werden. Aber das ist ein Thema, mit dem Sie sich in den Bereichen IT-Sicherheit und Betriebssysteme auseinander setzen.

10.1.9 Zusammenfassung

Obwohl wir jetzt nur acht Begriffe zur Verfügung haben können wir damit jede denkbare Programmiersprache beschreiben. Wir werden dabei zwar ggf. neue Begriffe einführen, um die Kombination aus diesen Begriffen mit einem einfachen Begriff zu bezeichnen, aber im Kern müssen Sie nur diese acht Begriffe vollständig verstehen, um zu verstehen, was Programmiersprachen unterscheidet:

- Bezeichner
- Pointer
- Wert
- Interpretation des Wertes
- Operationen
- Lebensdauer
- Zugriffsrecht

10.1.10 Literale

Literale sind keine Eigenschaft von virtuellen Obejekten, aber der Begriff wird Ihnen in vielen Programmiersprachen begegnen. Ein Literal ist vereinfacht ausgedrückt ein Zeichen, das in einer Programmiersprache benutzt werden kann. Das bedeutet, dass es häufig Zahlen, Buchstaben und andere Schriftzeichen sind, die Sie aus gesprochnen Sprachen kennen.

Wichtig:

- Es können aber auch Zeichen wie # sein, die Sie eher nicht als Schriftzeichen verstehen würden.
- Außerdem ist ein Literal, das Sie als Zahl kennen (z.B. die Ziffer 2) in einer Programmiersprache nicht unbedingt eine Zahl. Es kann dort auch durchaus so verwendet werden wie ein Buchstabe in einer gesprochenen Sprache.

Und weil es diese beiden Unterschiede zu nicht-Programmiersprachen gibt, wird bei Programmiersprachen des öfteren nicht von Schriftzeichen, Buchstaben, Zahlen oder ähnlichem gesprochen, sondern von Literalen. Wenn Sie also eine Programmiersprache erlernen, dann müssen Sie zunächst lernen, welche Literale in dieser Sprache wofür verwendet werden.

10.2 Virtuelle Objekte in Programmiersprachen

In imperativen Programmiersprachen gibt es verschiedene Arten virtueller Objekte. Die einfachsten virtuellen Objekte werden dabei üblicherweise als Konstanten und Variablen bezeichnet. Konstanten sind dabei Sonderformen von Variablen, die sich bei der Programmierung dadurch von Variablen unterscheiden, dass Ihr Wert nur einmal festgelegt, aber für den Rest der Lebensdauer einer Variablen nicht mehr geändert werden kann.

Funktionale Programmiersprachen zeichnen sich dadurch aus, **dass ALLE Variablen Konstanten sind**. **Java** ist seit der Version 8 ein Sonderfall: Hier können Sie weitgehend imperativ programmieren, also mit Variablen und Konstanten. Aber es gibt auch funktionale Anteile seit dieser Java-Version. In diesen sind Variablen dementsprechend immer Konstanten.

Die theoretische Grundlage zur funktionalen Programmierung nennt sich **Lambda-Kalkül**. Wenn Sie also in Programmierung über den Begriff Lambda (z.B. als Symbol für die Wellenlänge) stolpern und es nicht um physikalische Grundlagen geht, dann geht es meist um funktionale Programmierung. Das Lamda-Kalkül ist ein Ansatz, der Effizienz zum zentralen

Dreh- und Angelpunkt der Programmierung erklärt und es ist aus der Perspektive der imperativen Programmierung nicht zu verstehen. Aufgrund der strikten Fokussierung auf effizienten Code ist funktionale Programmierung meist nicht leicht lesbar. Dafür ist es ein Programmierstil, der mit extrem wenig Code komplexe Probleme löst. Der Einstieg in funktionale Programmierung fällt den meisten Programmierern also schwer. Häufig behaupten sie deshalb, es handle sich um einen ganz unsinnigen Ansatz zu programmieren. Wenn Sie (Medien-)InformatikerIn werden wollen, aber dennoch diese Aussage von sich geben, dann zeigt das, dass Sie in diesem Bereich inkompetent sind. Das bedeutet nicht, dass der Einstieg unbedingt leicht wäre, er erfordert wie der Einstieg in jede(!) Art der Programmierung Zeit und Anstrengung.

10.2.1 Primitive virtuelle Objekte

In verschiedenen Programmiersprachen werden Sie auf den Begriff des **Primitive** bzw. des primitiven Datentyps stoßen. Das sind virtuelle Objekte, die nicht mehr weiter aufteilbar sind. Wenn wir beispielsweise eine Variable haben, unter der ein Buchstabe oder eine Zahl gespeichert ist, dann ist das ein Primitive. Wenn dagegen eine Variable auf eine Funktion verweist oder ein Objekt wie in Java oder auf beliebige andere zusammengesetzte virtuelle Objekte (also auch Datenstrukturen), dann handelt es sich nicht um einen Primitive.

Leider wird der Begriff des Primitives meist nicht genutzt, obwohl er für eine differenzierte Betrachtung von Programmiersprachen sehr praktisch ist. Deshalb haben Einsteiger häufig ein Problem damit, dass eine Variable in typisierten Sprachen mal auf eine einzelne Zahl, mal auf einen Text und mal auf eine Datenstruktur verweisen kann. Um hier eine klare Unterscheidung zu treffen wird in diesem Buch der Begriff des **Datentyp**s ausschließlich für primitive Datentypen genutzt. Ebenso wird der Begriff **Variable** nur für virtuelle Objekte genutzt, die ein Primitive sind.

10.2.2 Deklaration einer Variablen

Wenn wir durch eine entsprechende Programmzeile eine Variable erzeugen, wird das als **Deklarierung** bezeichnet. Bei der Deklarierung wird bei jeder imperativen Programmiersprache ein Bezeichner festgelegt.

In **statisch typisierten** Sprachen wie C, C++ und Java wird außerdem bei der Deklarierung durch Programmierer der sogenannte Datentyp festgelegt.

Der **Datentyp** ist nichts anderes als die Interpretation des Wertes eines virtuellen Objekts, womit auch die für das virtuelle Objekt gültigen Operationen festgelegt werden. Zusätzlich wird ein Pointer angelegt. Außer bei der maschinennahen Programmierung passiert das durch die Programmiersprache. (Anm.: Wenn Sie es mit Programmiersprachen zu tun haben, die die manuelle und automatische Festlegung des Pointers bei der Deklaration einer Variablen erlauben, müssen Sie sicherstellen, dass Sie nicht versehentlich auf einen Speicherbereich referenzieren, der bereits durch ein anderes virtuelles Objekt genutzt wird. In diesem Teil des Buches werden wir damit aber nichts zu tun haben.)

Lassen Sie sich hier bitte nicht davon verwirren, dass die Interpretation des Wertes festgelegt wird, bevor ein Wert festgelegt wird. Der Grund ist recht simpel: Es wird hier nicht nur festgelegt, wie der Wert interpretiert werden soll, sondern auch wie viel Speicher für die Speicherung des Wertes verwendet werden soll.

Wenn Sie schon mit Java programmiert haben, dann können Sie jetzt vollständig nachvollziehen, warum es mehrere Datentypen für ganzzahlige Werte gibt.

Wenn eine Programmiersprache bei statisch typisierte Variablen eine Regelung für das Zugriffsrecht hat, dann muss dieses ebenfalls bei der Deklaration festgelegt werden. Damit haben Sie beispielsweise bei der klassenbasierten Objektorientierung wie in Java zu tun.

1. Bei statisch typisierten, kompilierten Sprachen mit Regelung des Zugriffs wie Java sieht die Deklaration einer Variablen so aus:

   ```
   zugriffsrecht datentyp bezeichner;
   ```

   Beachten Sie aber bitte, dass das Semikolon ein Zeichen ist, das in C, C++ und Java das Ende einer Programmzeile anzeigt. In anderen Sprachen werden andere Zeichen oder schlicht der Zeilenumbruch dafür genutzt.

2. Bei Java gibt es eine Besonderheit, wenn Sie eine Konstanate deklararieren wollen. In dem Fall müssen Sie zusätzlich das Schlüsselwort `final` nutzen:

   Wichtig: Es gibt in Java auch das Schlüsselwort `static`, aber das bedeutet gerade nicht statisch bzw. konstant. Wir werden erst bei der Einführung in die objektorientierte Programmierung mit Java darauf zurück kommen.

```
zugriffsrecht final datentyp bezeichner;
```

Beachten Sie aber bitte, dass das Semikolon ein Zeichen ist, das in C, C++ und Java das Ende einer Programmzeile anzeigt. In anderen Sprachen werden andere Zeichen oder schlicht der Zeilenumbruch dafür genutzt.

3. Bei statisch komplierten Sprachen ohne Zugriffsregelung wie C sieht die Deklaration einer Variablen so aus:

   ```
   datentyp bezeichner;
   ```

4. Bei dynamisch typisierten Sprachen gibt es in aller Regel keine Deklaration, weil die Programmiersprache im Moment der Wertzuweisung zu einem neuen Bezeichner automatisch einen Pointer und einen Datentyp in Abhängigkeit vom Wert des virtuellen Objekts festlegt.

5. Bei Sprachen, die virtuelle Obejkte ohne einen Wert nutzen gibt es ebenfalls keine Deklaration, sondern wir verwenden einen Bezeichner, ohne ihn vorher in irgend einer Weise eingeführt zu haben.

Zur Erinnerung:

Dynamisch typisierte Sprachen unterscheiden sich von **statisch typisierten** Sprachen dadurch, dass bei Ihnen der Datentyp von der Sprache selbst verwaltet wird. Streng genommen bedeutet dynamisch typisiert lediglich, dass sich der Datentyp einer Variable im Laufe des Programmablaufs ändern kann. Deshalb wird häufig bei statisch typisierten Sprachen von einer Typsicherheit gesprochen, aber der Begriff ist unsinnig. Denn dynamisch typisierte Sprachen ändern den Datentyp einer Variablen nach festen Regeln. Und ob nun Sie als ProgrammiererIn einen Datentyp festlegen oder ob das Programm das nach festen Regeln tut: In beiden Fällen hat eine Variable zu jedem Zeitpunkt einen bestimmten Datentyp. Der Unterschied ist der: Bei statisch typisierten Sprachen können Sie direkt im Programm nachlesen, welchen Datentyp ein virtuelles Objekt hat. Wobei das auch nur dann gilt, wenn der Entwickler nicht zu chaotisch programmiert hat. Bei dynamisch typisierten Sprachen müssen Sie dagegen die Regeln lernen, nach denen die Sprache den Datentyp ändert bzw. festlegt, um zu erkennen, wann eine Variable welchen Datentyp hat.

10.2.3 Initialisierung einer Variablen

Der zweite Schritt bei der Arbeit mit einer Variablen ist die **Intialisierung**. Hier wird der Wert der Variablen festgelegt. In vielen statisch typisierten

Sprachen kann eine Variable in einer Zeile deklariert und initialisiert werden. In einigen Sprachen müssen Sie dagegen alle Variablen zuerst explizit deklarieren, bevor Sie sie verwenden dürfen.

Die Initialisierung ist die erste Operation, die für Variablen (also virtuelle Objekte einer imperativen Programmiersprache) definiert ist. Im Gegensatz zu den meisten anderen Operationen ist diese also für alle Variablen unabhängig vom Datentyp gültig.

1. Bei statisch typisierten, kompilierten Sprachen wie C, C++ oder Java sieht die Deklaration einer Variablen so aus:

   ```
   bezeichner = wert;
   ```

2. Bei dynamisch typisierten Sprachen wie Ruby gibt es keine Deklaration und wir initialisieren direkt:

   ```
   bezeichner = wert;
   ```

3. PHP wählt hier einen Sonderweg, weil Variablen hier durch ein $-Zeichen ausgezeichnet werden. Aber ansonsten ist das System konsistent:

   ```
   $bezeichner = wert;
   ```

4. In PHP gibt es aber auch Variablen ohne führendes $-Zeichen. Das sind Konstanten, die mit dem Schlüsselwort `const` programmiert werden müssen:

   ```
   const bezeichner = wert;
   ```

5. Auch wenn es trivial erscheint, sei hier nochmal betont: Sprachen, die virtuelle Objekte ohne einen Wert kennen haben natürlich keine Initialisierung.

10.2.4 Kombination von Deklaration und Initialisierung

In vielen etwas weniger streng definierten Sprachen kann die Deklaration und die Intialisierung virtueller Objekte in einer Zeile kombiniert werden. Dementsprechend ergeben sich u.a. die folgenden Varianten:

1. `zugriffsrecht final datentyp bezeichner = wert;`
2. `zugriffsrecht datentyp bezeichner = wert;`
3. `datentyp bezeichner = wert;`

In Sprachen wie C und Pascal kann der „Datentyp“ einer Variablen ein Pointer sein. Das bedeutet, dass wir dort nicht etwa eine Zahl, einen Text oder ein anderes konkretes Objekt abspeichern, sondern einen Verweis auf den Wert einer anderen Variablen. Das ermöglicht einen sehr effizienten Programmierstil für bestimmte Arten von Problemen, die tatsächlich recht häufig auftreten. Ein vollständiges Verständnis dafür können Sie aber im Grunde erst dann entwickeln, wenn Sie die Grundlagen der maschinennahen Programmierung beherrschen.

In objektorientierten Sprachen wie Java und JavaScript können Sie wichtige Anwendungsbereiche von Pointern durch alternative Methoden umsetzen. Wir kommen darauf zu sprechen, wenn es um die Programmierung von eigenen Datenstrukturen in Java geht.

10.2.5 Zuordnung eines Wertes

Recht häufig werden Sie von der **Zuordnung eines Wertes zu einer Variablen** hören. Das ist nichts anderes als die Änderung des Wertes einer Variablen. Es bedeutet, dass der Wert einer Variablen geändert wird.

Um eine Wertzuordnung zu **programmieren** müssen Sie genau das selbe wie bei der Initialisierung einer Variablen machen.

Auch wenn es Ihnen vielleicht schon zu den Ohren herauskommt: Wenn eine Sprache keine Wert von virtuellen Objekten hat, dann können Sie einem virtuellen Objekt in dieser Sprache natürlich auch keinen Wert zuordnen.

10.2.6 Wert eines virtuellen Objekts

Ein häufiges Missverständnis von Einsteigern besteht darin, das Wort `Wert` mit einer Zahl gleichzusetzen. Das ist zwar insofern richtig als die meisten Computer mit Binärwerten arbeiten, aber es ist in sofern falsch als der Wert eines virtuellen Objekts (also auch der Wert einer Variablen) auch ein Buchstabe sein kann.

Sehr häufig programmieren wir keinen konkreten Wert (also eine Zahl oder einen Buchstaben), sondern eine Operation oder Funktion als Wert einer Variablen. Es gibt jetzt zwei Möglichkeiten:

- In den meisten Fällen bedeutet das, dass die Operation oder Funktion zunächst ausgewertet wird, und dass das „Ergebnis“ davon der Wert ist, der der Variablen zugeordnet wird.

- Es gibt aber noch eine zweite Möglichkeit, auf die wir an dieser Stelle noch nicht eingehen: Bestimmte Programmiersprachen erlauben es, dass eine Funktion unter einem Bezeichner gespeichert werden. Bei diesen Fällen wird eine solche Funktion als **first-class object** bezeichnet. Der Wert der Variablen ist dann nicht das Ergebnis der Funktion, sondern der Wert der Variablen ist die Funktion selbst. Momentan können Sie damit noch nichts anfangen, aber es ist wichtig, dass Ihnen klar wird, dass eine Funktion beim Programmieren nicht nur etwas ist, dass wir direkt „ausrechnen“ lassen können.

Wenn wir uns mit Konzepten wie der der objektorientierten Programmierung beschäftigen, dann kann der Wert eines virtuellen Objektes auch eine Sammlung von anderen virtuellen Objekten, Verweise auf Dateien oder sogar zu vollständigen Programmen sein. Ein Beispiel dafür haben Sie gerade gesehen: In manchen Programmiersprachen ist der Wert einer Variable der Verweis auf eine Funktion unseres Programms.

10.2.7 Anonyme virtuelle Objekte

Stellen Sie sich für diesen Abschnitt vor, wir hätten einer Variable a, der wir den Wert eine Operation zugeordnet haben. (Sprich, wir haben eine Programmzeile wie `a = y + z;`.)

Wenn der Rechner eine Operation (in unserem Beispiel die Summe von y und z) einer Programmiersprache auswertet, dann bleiben die Werte, die für diese Operation verwendet wurden (y, z) in aller Regel erhalten und werden nicht überschrieben. Auch das Ergebnis bleibt zumindest für eine kurze Zeit erhalten. Das bedeutet aber auch, dass das Ergebnis einer Operation irgendwo gespeichert werden muss.

Vielleicht denken Sie jetzt, dass das Ergebnis doch sofort als Wert der Variablen a zugeordnet wird, aber auf unterster Ebene steht das nicht eindeutig fest.

So trivial das klingt führt es also zu der Frage: Wo wird dieses Ergebnis gespeichert? Wenn eine Operation ausgewertet wird, haben wir ja noch keine Variable deklariert, die das Ergebnis als Wert zugeordnet bekommt. Ansonsten müsste der Rechner ja abstürzen, wenn eine Operation ausgeführt wird, ohne dass das Ergebnis einer Variablen zugeordnet wird.

Und das führt direkt zum Begriff der **anonymen Variable**. Eine anonyme Variable ist also eine Variable, die einen Wert, einen Datentyp und einen Pointer aber noch keinen Bezeichner hat. Es gibt also einen Binärwert, der in einer Speicheradresse gespeichert ist und der von der Programmierspra-

che als ein Wert im Sinne eines Datentyps interpretiert wird.

Natürlich besteht der nächste Schritt darin, dass dieser Wert einer Variablen zugeordnet wird, aber zuvor wird er wie beschrieben in einem Speicherbereich abgelegt. Es gibt danach also zwei Möglichkeiten:

1. Entweder wird nun der Wert der (im Beispiel mit a bezeichneten) Variable mit dem Wert der anonymen Variable überschrieben.
2. Oder der Pointer dieser Variablen wird auf die Adresse der anonymen Variablen geändert.

Aufgabe

Sie wissen jetzt, was eine Zuordnung ist. Sie wissen auch, was eine anonyme Variable ist. Überlegen Sie sich nun, warum nach der Zuordnung des Wertes einer anonymen Variablen zu einer deklarierten Variable Speicher verschwendet wird. Wenn Ihnen das klar ist, recherchieren Sie, was der sogenannte **Garbage Collector** ist.

10.2.8 Schreibweise von Bezeichnern

Je nach verwendeter Programmiersprache gibt es unterschiedliche Konventionen (Vereinbarungen) darüber, wie Bezeichner geschrieben werden. Diese unterscheiden sich von Sprache zu Sprache und müssen erlernt werden; es gibt keine logische Begründung, aus der sich die Konvention ergibt.

10.3 Datentypen

Sie wissen bereits, dass der Datentyp bei statisch typisierten Sprachen genau wie bei dynamisch typisierten Sprachen festlegt, wie die Programmiersprache den Wert einer Variablen interpretiert. Wenn Sie sich nun die Datentypen für ganze Zahlen bei Java ansehen, dann werden Sie feststellen, dass es dafür verschiedene Datentypen gibt:

1. `byte`
2. `short`
3. `int`
4. `long`

Wenn also der Datentyp nur festlegen würde, wie die Werte einer Variablen interpretiert werden, dann würde eine Definition von vier verschiedenen Variablen, die alle ausschließlich ganzzahlige Werte speichern können

keinen Sinn machen: Es sind alles ganze Zahlen, also müsste doch ein Datentyp reichen.

Tatsächlich gibt der Datentyp einer Variablen nicht nur vor, wie ein gespeicherter Binärwert an einer Addresse des Speichers interpretiert werden soll, sondern er gibt auch an, wie viele Bit für den Wert verwendet werden. Wenn Sie sich später mit maschinennaher Programmierung beschäftigen, wird Ihnen das vollständig klar werden. Für den Moment sei nur gesagt:

1. Eine `byte`-Variable in Java belegt ein Byte (also 8 Bit) im Speicher. Das entspricht einem Intervall von -128 bis 127.

2. Eine `short`-Variable in Java belegt zwei Byte (also 16 Bit) im Speicher. Das entspricht einem Intervall im Bereich von $-3,2 \cdot 10^4$ bis $3,2 \cdot 10^4$.

3. Eine `int`-Variable in Java belegt vier Byte (also 64 Bit) im Speicher. Das entspricht einem Intervall im Bereich von $-2,2 \cdot 10^9$ bis $2,2 \cdot 10^9$.

4. Und eine `long`-Variable in Java belegt acht Byte (also 128 Bit) im Speicher. Das entspricht einem Intervall im Bereich von $-9,2 \cdot 10^18$ bis $9,2 \cdot 10^18$.

Wenn Sie also eine Highscore-Liste programmieren wollen und dafür eine Variable vom Typ int programmieren, dann sollten Sie sicher sein, dass die gespeicherten Werte nicht über 30 Mrd. steigen können. Im besten Falle erhalten Spieler sonst anstelle einer neuen Highscore plötzlich Werte im Bereich von -30 Mrd. Punkten, im schlimmsten Fall stürzt Ihr Programm ab.

Aufgaben

1. Mit dem, was Sie bis jetzt über Datentypen wissen, müssen Sie die folgende Frage beantworten können: Warum kann `String` kein Datentyp sein?

2. Der Datentyp einer Variablen ist in C statisch. Das heißt, er kann nicht geändert werden. Überlegen Sie sich, was Sie tun könnten, wenn Sie den Datentyp einer Variablen ändern wollen, damit Sie mit der Variablen weiterarbeiten können.
   Hinweis: Es ist unmöglich den Datentyp einer Variablen zu ändern. Also muss es eine andere Möglichkeit geben.

Die Änderung des Datentyps einer Variablen wird als **Typecasting** bezeichnet.

10.3.1 Datentypen in C und Java

Es gibt noch mehr als die hier aufgeführten Datentypen, aber selbst für Fortgeschrittene sollten diese genügen.

Bitte beachten Sie, dass Groß- und Kleinbuchstaben aber auch Zeichen wie der Unterstrich fest zugeordnet sind. Die drei Bezeichner `bool`, `Bool` und `\_Bool` sind also weder das Selbe noch das Gleiche.

10.3.2 Datentypen für ganze Zahlen

Ähnlich wie bei Java gibt in diesem Bereich in C vier Datentypen, die danach unterteilt werden, wie viel Bit an Speicherplatz sie belegen. Leider ist die Bedeutung nicht identisch mit der in Java.

1. `short` hat in C 16 Bit Länge.
2. `int` und `long` haben in C 32 Bit Länge.
3. `long long` (nicht mit dem einfachen long verwechseln!) hat in C 64 Bit Länge.

Hier zum Vergleich nochmal die Datentypen in Java. Sie werden alle mit dem Wert 0 initialisiert. Eine explizite Initialisierung ist also nicht nötig:

1. `byte` in Java 8 Bit: -128 bis 127.
2. `short` in Java 16 Bit: $-3,2 \cdot 10^4$ bis $3,2 \cdot 10^4$.
3. `int` in Java 32 Bit: $-2,2 \cdot 10^9$ bis $2,2 \cdot 10^9$.
4. `long` in Java 64 Bit $-9,2 \cdot 10^18$ bis $9,2 \cdot 10^18$.

Damit sind die einzigen ganzzahlige Datentypen, den Sie bei beiden Sprachen gleich nutzen können `short` und `int`. Wollen Sie dagegen besonders große Zahlen verwenden, müssen Sie differenzieren, ob Sie ein einfaches oder doppeltes long programmieren müssen.
Programmierbeispiele

Wenn Sie die Variable `punktestand` für ein Fußballspiel programmieren wollen, dann lautet in Java die Deklaration und Intialisierung:

```
int puntestand;
```
Hier könnten Sie zwei Dinge verwundern:

1. Wenn Sie sich wundern, warum hier kein Wert zugeordnet wurde, dann lesen sie nochmal etwas weiter oben nach, mit welchem Wert ganzzahlige Variablen in Java immer initilisiert werden, außer wenn das vom Programmierer explizit überschrieben wird.

2. Wenn Sie sich wundern, warum da kein Schlüsselwort für die Zugriffsbeschränkung steht, dann kommt hier ein kleines aber feines Detail: Es gibt eine Beschränkung des Zugriffs, für die es kein Schlüsselwort gibt. Sie werden später noch feststellen, dass die Feinheiten der Zugriffsbeschränkung in Java nicht systematisch sind, aber das ist bis zum Beginn der objektorientierten Programmierung irrelevant.

10.3.3 Operationen für ganzzahlige Variablen

Die Operationen `+`, `-`, `*` stehen für Addition, Subtraktion und Multiplikation und können problemlos angewendet werden. Das Ergebnis ist eine ganzzahlige Variable und so lange wir auf den Wertebereich achten, können wir zwei `int`-Variablen problemlos mit einer dieser Operationen verknüpfen: Das Ergebnis wird wieder eine `int`-Veriable sein.

Doch was ist mit der Division (programmiert als `\`)? Die ist tatsächlich ein Problem, denn abgesehen von `division by zero` gibt es hier ja noch Unklarheiten, wenn das Ergebnis eine ganzrationale Zahl ist: In statisch typisierten Sprachen können Sie davon ausgehen, dass eine Division zweier ganzer Zahlen immer ein ganzzahliges Ergebnis ist, bei dem die Stellen nach dem Komma schlicht unter den Tisch fallen. Also ergibt `1 / 4 = 0` und nicht `1 / 4 = 0.25`!

Allerdings müssen Sie hier bei jeder Programmiersprache wieder genau nachschlagen, wie ein solcher Fall ausgewertet wird und sich genau ansehen, was die Auswirkung davon ist. (Anm.: Es ist schon faszinierend, dass die Anhänger von statisch typisierten Sprachen darin kein Unsicherheitsproblem sehen, schließlich ist 1 : 4 nicht gleich 0.)

Andererseits habe ich noch einen Operator unter der Tisch fallen lassen, mit dem Sie sich in solchen Fällen absichern können: Es geht um den Modulo-Operator, der als `%` programmiert wird. Die Sicherheit erreichen Sie in dem Fall z.B. wie folgt:

```
x = a / b;
y = a % b;
```

Jetzt müssten Sie nur noch prüfen, ob y gleich 0 ist, um sicher zu gehen, dass „nichts unter den Tisch fällt“. Genau das ist aber der zentrale Nachteil von streng typisierten Sprachen gegenüber dynamisch typisierten Spra-

chen: In den meisten Fällen müssen Sie deutlich mehr Zeilen programmieren, ohne davon einen Mehrwert zu haben.

Hier nochmal die arithmetischen Operationen für ganzzahlige Variablen:

- `+` Addition
- `-` Subtraktion
- `*` Multiplikation
- `\` Division
- `%` Modulo

Es gibt noch eine Reihe weiterer Operationen, die für ganzzahlige Variablen definiert sind, aber dazu kommen wir nach der Einführung weiterer Datentypen, da das Ergebnis dieser Operationen ein Wert mit einem anderen Datentyp ist.

10.4 Das erste Programm

Wenn wir hier mit interpretierten Sprachen angefangen hätten, dann hätten wir die Programmzeilen direkt in den Interpreter eintragen können und dieser hätte umgehen die Ergebnisse ausgegeben. Bei C (bzw. C++) sowie Java haben wir es aber mit kompilierten Sprachen zu tun. Also besteht der erste Schritt darin, den Quellcode in einer Datei abzuspeichern, ihn zu kompilieren und die neue Datei zu starten. (Keine Sorge, das machen wir gleich Schritt für Schritt.)

Zur Wiederholung: Programme einer kompilierten Sprache werden vollständig in ein Format umgewandelt, das direkt vom Betriebssystem bzw. der Laufzeitumgebung der Programmiersprache ausgeführt werden kann. Diese Umwandlung wird als Kompilieren bezeichnet.

10.4.1 Kleines Programm in C

Wenn Sie entsprechend des Kapitels zur Vorbereitung des Rechners GCC installiert haben, dann können Sie direkt einen Editor öffnen und folgendes kurzes Programm eingeben. Speichern Sie es dann (idealerweise in einem Verzeichnis wie `C:\a_meine_programme`) unter dem Namen `helloMarvin.c`):

```
#include <stdio.h>
main()
```

```
{
int a = 42;
int b = 2 * a;
printf("Die Antwort lautet: " + b );
}
```

Wechseln Sie jetzt auf der Konsole in das Verzeichnis `C:\a_meine_programme` und geben Sie dort `gcc helloMarvin.c` ein und anschließend `a`. Darauf wird eines der unter InformatikerInnen beliebtesten Zitate ausgegeben.

Was die include-Anweisung, das main() oder die geschweiften Klammern bedeuten schauen wir uns im Anschluss an die Besprechung der Datentypen an, aber vorher schauen wir und die gleiche Aufgabe in Java an:

10.4.2 Kleines Programm in Java

```
public class HelloMarvin{
public static void main(String[] args)
{
int a = 42;
int b = 2 * a;
System.out.println("Die Antwort lautet: " + b );
}
}
```

Nachdem Sie diese Datei im Verzeichnis `C:\a_meine_programme` unter dem Namen `HelloMarvin.java` gespeichert haben, geben Sie `javac HelloMarvin` ein und anschließend `java HelloMarvin`.

Wie Sie sehen sind die Unterschiede zwischen beiden Programmen gar nicht so groß: Das eigentliche Programm ist bis auf ein Detail identisch. Nur drum herum steht eine ganze Menge Code, der Ihnen momentan unklar sein dürfte.

10.4.3 Kleines Programm in Pascal

```
program HelloMarvin(output);
var
a : integer;
b : integer;
begin{main}
a := 42;
b := 2 * a;
```

```
writeln('Die Antwort lautet: ' + b);
end{main}.
```

Aufgabe:

Vergleichen Sie diesen Code mit dem in C und Java. Was ist der zentrale Unterschied bei der Programmierung der Variablen? (Nein, es ist nicht die Tatsache, dass der Datentyp hier `integer` und bei den beiden anderen `int` lautet.)

10.4.4 Kleines Programm in PHP

PHP kennen Sie ja bereits aus dem ersten Teil des Buches:

```
$a = 42;
$b = 2 * $a;
echo("Die Antwort lautet: " . b);
```

Aufgabe:

Selbst wenn Sie es nicht mehr wissen, sollten Sie jetzt sofort einen Unterschied zwischen PHP einerseits und C, C++ sowie Java andererseits benennen können. Welcher ist das?

10.4.5 Kleines Programm in Scheme

Scheme ist ein Lisp-Dialekt und damit eine funktionale Sprache. Funktionale Sprachen sind wie beschrieben dafür gedacht, so effizient wie möglich zu programmieren.

```
(define a 42)
(define b (* 2 a))
(display "Die Antwort lautet: " b)
```

Aufgabe:

Beschreiben Sie, welche Unterschiede Sie zwischen der funktionalen Programmierung und der imperativen Programmierung anhand dieses kleinen Beispiels erkannt haben.

10.4.6 Kein sinnvolles Problem für PROLOG

Die Teile unserer Programme, die wir in PROLOG umsetzen können, habe ich hier zusammengefasst. Wie Sie sehen fehlt da die Textausgabe. Aber

das ist kein Wunder: PROLOG ist eine Sprache für logische Programmierung, das bedeutet, das PROLOG nicht dafür gedacht ist, um nett formulierte Ausgaben durchzuführen, sondern einzig dafür, aus einer komplexen Menge an Fakten mögliche Lösungen zu ermitteln. Wenn wir also unsere kleine Aufgabe in PROLOG umsetzen, dann ist das, als wenn wir einen LKW nutzen, um eine Handvoll Staub aus der Wohnung zum Mülleimer zu transportieren. Die folgenden zwei Zeilen sind in PROLOG zwar möglich aber sowohl stilistisch schlecht als auch inhaltlich weitgehend sinnfrei.

```
a is 42.
b is 2 * a.
```

Wir werden uns in einem späteren Kapitel ansehen, was es mit logischer Programmierung auf sich hat und dann werden Sie verstehen, warum dieses PROLOG-Programm so unsinnig ist.

Jetzt aber wieder zurück zu Datentypen in Programmiersprachen. Wir bleiben hier jedoch vorerst bei rein imperativen Programmiersprachen:

10.5 Fortsetzung zu Datentypen

Die Tatsache, dass der eigentliche Kern nahezu gleich ist, ist der Grund, aus dem ein Umstieg von Java auf C oder umgekehrt gar nicht so schwer ist. Das wiederum ist auch der Grund, warum ich hier beide Sprachen gemeinsam vermittle: Da beide Sprachen sehr gebräuchlich ist, ist es gut, wenn Sie beide beherrschen. Die größte Schwierigkeit beim Umstieg von Java zu C bzw. C++ ist die Pointerarithmetik und umgekehrt tun sich die meisten C bzw. C++ ProgrammiererInnnen schwer damit, darauf zu verzichten. Dabei bietet Java Ihnen da eine sehr angenehme Alternative an. Doch zu diesen Details später mehr.

Doch bevor wir uns weiter mit Datentypen auseinander setzen, kommen wir zu einem weiteren Thema, bei dem Sie ausnahmsweise auswendig lernen müssen und das für jede einzelne Programmiersprache aufs neue:

10.5.1 Literale und reservierte Schlüsselwörter

Oben haben Sie einfach Programmbeispiele abgetippt. Dabei haben Sie am Anfang dieses Kapitels gelernt, dass es für jede Programmiersprache zugelassene Literale und bei der Wahl der Bezeichner feste Regeln gibt, wie aus diesen Literalen die Bezeichner von virtuellen Objekten formuliert werden dürfen.

Literale in C

Literale in Java

Literale in PHP

10.5.2 Datentypen für Buchstaben

Sie mögen sich wundern, warum jetzt nicht die Datentypen für Fließkommazahlen folgen, aber das ist recht simpel: Für Rückmeldungen werden Sie im Regelfall Texte (also aus Buchstaben zusammengesetzte Ausgaben) benötigen. In den C, C++ und Java heißt der Datentyp zwar immer `char`, aber in C und C++ hat er nur eine Länge von 8 Bit, während ein `char` in Java 16 Bit belegt.

Damit haben wir auch hier wieder eine Situation, in der der gleiche Datentyp in nahe verwandten Programmiersprachen unterschiedlich umgesetzt wird: In C und C++ können wir in einer `char`-Variablen Symbole aus der ASCII-Codierung speichern (also beispielsweise keine Umlaute), während wir in einer `char`-Variablen in Java Symbole entsprechend der UTF-16-Codierung speichern und damit ohne weiteren Aufwand praktisch alle Zeichen von heute gesprochenen Sprachen nutzen können.
Kontrolle

Deklarierung und Initialisierung von char-Variablen:

char einBuchstabe = ‘a‘;
Wichtig: Im Gegensatz zu Strings, die wir uns erst im Rahmen der Datenstrukturen ansehen werden beginnen und enden char-Variablen mit einem einfachen Anführungszeichen.

10.5.3 Datentypen für Wahrheitswerte

Richtig gelesen: Es gibt einen Datentyp für Variablen, die nur unterscheiden, ob etwas wahr oder falsch ist. Solche Variablen nennt man auch boolesche Variablen. Den meisten Programmierern ist gar nicht klar, wie oft Sie mit diesen Variablen arbeiten, weil sie meist durch eine Operation erzeugt werden, deren Ergebnis sofort und danach nie wieder genutzt wird. Und in diesem Fall brauchen Sie keine Variable dieses Datentyps deklarieren.

Da das komplizierter klingt, als es ist, hier ein Beispiel: Sie können innerhalb eines Programms einfache wenn-dann-Abfragen wie die folgende programmieren: Wenn es kälter als 12 Grad ist, zieh dir einen Pullover an, sonst lass das. Hierzu wird ein Vergleich einprogrammiert, bei dem eine

Variable mit dem Wert 12 vergleichen wird. Das Ergebnis ist eine boolesche Variable. Wenn Sie aber wie in diesem Beispiel direkt die Auswirkung programmieren (... dann zieh dir einen/keinen Pullover an), dann brauchen Sie das Ergebnis der Vergleichsoperation keiner expliziten Variablen zuordnen.

Boolesche Variablen können Sie übrigens nur dann deklarieren, wenn sie die Headerdatei stdbool.h am Programmanfang includen. Der Datentyp lautet dann bool.

10.5.4 Datentypen für Fließkommazahlen

Zur Erinnerung: Nutzen Sie wann immer möglich ganzzahlige Datentypen, wenn Sie programmieren, da Sie durch die Zahlen zur Basis 2 bei Divisionen von Zahlen zur Basis 10 schnell Rundungsfehler bzw. unschöne Zahlendarstellungen bekommen.

In diesem Bereich gibt es lediglich drei Datentypen, wobei zwei davon wieder identische Zahlenbereiche abbilden. Schlagen Sie diese Zahlenbereiche bei Bedarf nach.

• float hat 32 Bit Länge • double und long double haben 64 Bit Länge

10.5.5 Datentypen für aufzählbare Elemente

Diese Datentypen sind recht spannend, weil sie es Ihnen ermöglichen, mit wenig Speicherbedarf relativ große Datenmengen abzubilden. Sie haben bei den bisherigen Datentypen jeweils den Fall gesehen, wo Sie einer Variablen einen konkreten Wert direkt zuordnen konnten. Jetzt lernen Sie einen Datentyp kennen, mit dem Sie einer Variablen einen Wert aus einer List indirekt zuordnen können.

Nehmen wir an, Sie haben eine Liste von Farben (wie auch immer Sie diese programmieren können). Und Sie möchten jetzt die Möglichkeit haben, dass eine Variable nur Einträge aus dieser Liste als Wert beinhalten kann. In diesem Fall lautet der Datentyp:

• enum

10.5.6 Datentypen für Pointer

Pointer sind Datentypen, die nicht etwa den Inhalt einer Speicheradresse im Sinne von Zahlen, Buchstaben oder ähnlichem interpretieren, was in einer Codierungstabelle definiert wird, sondern die den Inhalt einer Speicheradresse als Speicheradresse verstehen.

Wenn Sie also einen Pointer haben, unter dessen Adresse der Wert 0x30 gespeichert ist, dann wird dieser Wert als die Adresse 0x30 „interpretiert". Im Grunde sind Pointer (und die zugehörige Speicherarithmetik) also eine Möglichkeit, all die Speicherzugriffe zu realisieren, die wir sonst nur bei Assembler durchführen könnten. Mit Pointern können wir beispielsweise all die Datenstrukturen realisieren, die C selbst nicht mitbringt.

Allerdings werden wir uns im Rahmen dieser Veranstaltung nicht mit Pointern beschäftigen; wenn Sie mit C oder C++ nach diesem Kurs weiter arbeiten wollen, müssen Sie das noch nachholen. Bei Java dagegen gibt es keine Pointer, weil dort der direkte Zugriff auf den Speicher ausgeschlossen ist.
Wichtig: Hier den PHP-Teil und den C-Teil sowie den imperativen Java-Teil integrieren.
Wichtig:

Alle Hinweise, die sich in diesem Abschnitt auf die Nutzung von Windows beziehen, gelten so für Windows 7. Bitte prüfen Sie ggf. selbständig, wie Sie unter 8 oder 10 vorgehen müssen. Allerdings sollten die Befehle in der Konsole auch dort bis auf die Verzeichnisstruktur von Windows vollständig identisch sein.

Das folgende ist für Windows-User, alle anderen nutzen bitte die entsprechenden Befehle und Programme ihres Betriebssystems. Um mit dem Programmieren zu beginnen, öffnen Sie bitte die sogenannte Eingabeaufforderung. (Bei anderen Betriebssystemen Console, Bash o.ä. genannt.)

Wenn bei den folgenden Aufgaben (Verzeichnis erstellen/ins Verzeichnis wechseln) Fehler passieren, dann haben Sie entweder keine ausreichenden Rechte, um diese Aufgabe an dem Rechner durchzuführen an dem Sie gerade sitzen oder Sie haben sich schlicht vertippt.

Mit dem Befehl mkdir C:/a_prog erstellen Sie bitte in der Eingabeaufforderung ein Verzeichnis. (Sie können es auch anders nennen, aber wenn Sie es so benennen, wird es am Anfang Ihrer Verzeichnisübersicht auftauchen.)

Wechseln Sie jetzt mit dem Befehl cd C:/a_prog in dieses neue Verzeichnis.

Starten Sie jetzt den Editor, den Sie im Bereich „Alle Programme“ unter Zubehör finden. Alternativ können Sie auch gerne einen einfachen Editor wie Notepad++ nutzen, der so wenige Komfortfunktionen bietet, dass er Einsteiger unterstützt anstatt Sie (wie die meisten IDEs) zu verwirren.

Hier ein kurzer C-Quellcode, den Sie bitte mittels des Editors im oben genannten Verzeichnis abspeichern. Nennen Sie das Programm beim Speichern am besten antwort42.c (wichtig ist nur, dass die Dateiendung .c ist) Dieser Code hat noch nichts mit der Programmierung des ARM-Prozessors zu tun, Sie können ihn also auf jedem Rechner programmieren und ausführen, der einen C-Compiler installiert hat.

```
\#include <stdio.h>

main()
\{
printf("Die Antwort lautet: 42");
\}
```

C-Programm, das die Zeile „Die Antwort lautet: 42“ (ohne Anführungszeichen) auf der Konsole ausgibt.

Wichtig: Wenn Sie unbedingt mit einer IDE beginnen wollen, dann wird dieser Code wahrscheinlich nicht genügen und Sie werden zunächst z.B. ein Projekt erstellen u.ä. Da es hier jedoch mehr als genug Fehlerquellen gibt, aus denen ein Programm nicht läuft, rate ich Ihnen an dieser Stelle von der Nutzung einer IDE ab. (Diejenigen, die eine IDE nutzen werden einige zusätzliche Aufgaben in diesem Text finden. Diese Aufgaben dienen dazu, dass Sie die IDE tatsächlich nutzen und sich langfristig einen professionellen Stil angewöhnen.)

Wenn Sie das ignorieren, sollten Sie eine erweiterte Fassung des Quellcodes verwenden wie diejenige, die Sie im Abschnitt zur Deklaration und Initialisierung von Variablen (folgt weiter unten) finden können.

10.6 Inhalt eines einfachen C-Programms

Unser C-Programm besteht aus drei Teilen:

• Zunächst wird eine sogenannte Headerdatei eingefügt. Was das im Detail bedeutet und wie es den Kompilierungsprozess beeinflusst, ist für das Verständnis des Programms nicht wichtig. Wie schon in den einleitenden

Kapiteln erläutert nutzen wir bei der Entwicklung von Programmen in aller Regel Teile, die von anderen Entwicklern programmiert wurden. Und Header-Dateien sind ein Beispiel für solchen wiederverwendeten Code.

Ein Hinweis bezüglich des Namens: stdio steht schlicht für standard input out. Damit ist klar, was für Aufgaben diese Headerdatei in unser Programm einführt: Sie ermöglicht es uns unter anderem, Ausgaben auf dem Bildschirm zu erzeugen und Eingaben von der Tastatur anzunehmen.

(Hinweis für die Programmierung in Java: Wenn Sie in Java so etwas wie System.out.println() verwenden, dann greifen Sie auf stdout zu. Wenn Sie in Java die sogenannte Scanner-Klasse nutzen, dann verwenden Sie meist stdin. Dabei werden Sie in Java nicht die Bezeichnungen stdout und stdin verwenden, aber das was Sie benutzen ist genau das gleiche. Und stdio kombiniert schlicht stdin und stdout.)

• Dann folgt die Funktion main(). Was Funktionen einer (imperativen) Programmiersprache sind, folgt im Abschnitt Operationen und Funktionen. Für den Moment sollten Sie sich lediglich merken, dass bei C, C++ und Java der Anfang eines Programms in der main()-Funktion steht.

• Nach main() folgen drei Zeilen, wobei die erste und letzte schlicht eine öffnende und schließende Klammer enthält. Diese beiden Klammern besagen schlicht, dass alles, was sich zwischen Ihnen befindet Teil der main() ist. So wie eine Variable in der Mathematik als Stellvertreter für einen Wert verwendet werden kann, kann ein Funktionsname (wie hier main) für einen Reihe von Befehlen verwendet werden.

• Damit bleibt nur noch die Frage, was denn diese Zeile tut, die mit printf beginnt. Das wird Teil mehrerer Aufgaben, die Sie demnächst lösen.

Kontrolle

Für diesen und einige der nachfolgenden Abschnitte gibt es keine explizite Kontrolle. Prüfen Sie bitte selbständig, ob Sie die Inhalte verstanden und umgesetzt haben. Beachten Sie bitte dabei, dass dieser Kurs für Einsteiger ohne Vorkenntnisse in der Programmierung erstellt wurde; Fortgeschrittene mögen bitte dennoch diese Abschnitte lesen, um festzustellen, wo ihre Erfahrungen von dem Abweichen, was hier beschrieben ist. Überlegen Sie dann bitte genau, worin diese Abweichung besteht, also ob es eine Vereinfachung für Einsteiger ist oder ob Sie sich unter einem Konzept tatsächlich etwas anderes vorstellen als das, was hier beschrieben ist. Beides ist selbstverständlich möglich.

10.6.1 Ihr erstes C-Programm: Vom Quellcode zum Program Image

Geben Sie jetzt in der Eingabeaufforderung den Befehl dir (bei anderen Betriebssystemen ls) ein, um zu sehen, welche Dateien sich in diesem Verzeichnis befinden. Sie sollten jetzt eine kurze Tabelle sehen, bei der unter anderem der Dateiname antwort42.c aufgeführt ist. Ist das nicht der Fall, dann haben Sie entweder den Quellcode gar nicht oder an einem anderen Ort abgespeichert.

Wie Sie wissen haben Sie damit Quellcode erzeugt, der noch nichts tut. Wechseln Sie jetzt bitte in die Eingabeaufforderung und geben Sie dort den Befehl gcc antwort42.c ein, um den Quellcode zu kompilieren.

Geben Sie nun nochmal dir (bzw. bei Linux ls) in der Eingabeaufforderung ein. Wie Sie sehen, befindet sich jetzt eine zweite Datei im Verzeichnis mit dem Namen a.exe . Links neben den Dateinamen befindet sich jeweils eine Zahl, die angibt, wie viele Byte jede Datei belegt.

Kontrolle

Warum hat unser Quellcode nur einen Umfang von 60 Byte, aber das Program Image hat mehr als 60.000 Byte? Woher kommt all dieser zusätzliche Inhalt?

10.6.2 Über die Nutzung des gcc Compilers

Wenn Sie nun den Buchstaben a eingeben und Enter drücken, passiert folgendes: Das Programm gibt den Satz Die Antwort lautet: 42 aus.

Frage: Warum heißt die Datei nicht antwort42.exe?

Die Antwort ist wieder einmal ganz simpel: Weil wir dem Compiler nicht gesagt haben, dass das kompilierte Programm unter dem Namen antwort42.exe gespeichert werden soll.

Frage: Was müssen wir tun, um einen Namen für die Ausgabedatei zu vergeben?

Wie so oft beim Verwenden von Programmen, die andere erstellt haben, müssen Sie jetzt suchen. GCC ist der Compiler, den wir verwenden und wenn Sie kein Linux-Nutzer sind, dann müssen Sie im Netz nach GCC, besser noch GCC HELP suchen. Ohne Englisch wird's jetzt schwer, aber das ist in der Informatik immer so, also entweder lernen Sie die Sprache spätestens jetzt oder Sie werden dauerhaft massiv benachteiligt sein. Kon-

kret suchen Sie nach so etwas wie einem gcc manual bzw. den gcc command options.

Kontrolle

(1) Finden sie heraus, wie Sie den gcc dazu bringen können, Ihre Datei unter dem Namen antwort42.exe zu speichern. (Nein, es reicht nicht aus, wenn Sie die Datei einfach selbst umbenennen, nachdem sie vom gcc kompiliert wurde.)

(2) Warum sollte der Dateiname nicht antwort 42.exe (also mit einer Leerstelle zwischen antwort und 42) heißen?

(3) Angenommen Sie wollen dennoch einen solchen Dateinamen erhalten. Was müssen Sie tun, um das zu realisieren?

10.6.3 Variablen

Was die Register für die maschinennahe Programmierung sind, sind die Variablen für die imperative Programmierung. Deshalb beginnen wir die Einführung in die imperative Programmierung mit der Einführung von Variablen.

Wichtig:

Wie schon bei der Einführung in die Programmierung in PHP ist es wichtig, dass Sie an dieser Stelle verstehen, dass Variablen nur eine Möglichkeit sind, um Daten im Rahmen einer Programmiersprache zu verwalten. Die Programmierung mit Hilfe von Variablen ist dabei ein zentrales Kennzeichen aller imperativen Programmiersprachen, doch wie gesagt gibt es noch wesentlich mehr Paradigmen als nur das imperative.

Das folgende gilt so nur für kompilierte Sprachen (also u.a. für C, C++ und Java). Bei interpretierten Sprachen gelten zwar zum Teil dieselben Grundlagen, aber wie schon in den einführenden Kapiteln beschrieben legen wir dort den Datentyp einer Variablen nicht durch einen „Befehl“ fest, sondern der Interpreter der Sprache legt selbständig einen Datentyp fest bzw. ändert diesen dynamisch und wir können ihn im Regelfall nicht einsehen.

Eine Variable ist schlicht ein Bezeichner, über den wir auf einen Wert bzw. ein Zeichen verweisen können. Im Gegensatz zu einem Register haben wir hier aber große Freiheiten, wenn es darum geht, wie eine Variable bezeichnet werden soll. Wenn Sie also beispielsweise eine Variable einrichten wol-

len, um eine zurückgelegte Strecke zu speichern, dann können Sie diese Variable zurueckgelegteStrecke nennen und brauchen nicht auf kryptische Bezeichnungen wie R12 zurück zu greifen, so wie das bei der Nutzung der Register war. Es gibt zwar gewisse Einschränkungen, welche Zeichen Sie benutzen dürfen, aber so lange Sie nur Buchstaben verwenden, die auch im englischen Alphabet vorkommen, sollten Sie kein Problem haben. Bis auf das erste Zeichen können Sie auch Zahlen und bestimmte Sonderzeichen verwenden.

Das Arbeiten mit Variablen ist vor allem deshalb ein mächtiges Werkzeug, weil wir hier so tun können, als wenn der Computer doch mehr könnte als nur Zahlen aus dem Speicher zu laden, sie zu addieren und sie wieder im Speicher abzulegen. Diese Arbeitsschritte brauchen wir dann nicht mehr zu programmieren; sie werden quasi von der Programmiersprache (genauer gesagt von der Laufzeitumgebung oder dem Interpreter der Programmiersprache) erledigt.

Wie Sie jetzt wissen speichern Sie streng genommen keinen Wert in einer Variablen, sondern der Wert wird immer noch unter einer Speicheradresse abgelegt. Dennoch sprechen wir bei imperativen Sprachen davon, dass wir einer Variablen einen Wert zuordnen, dass wir unter einer Variablen einen Wert speichern, oder dass wir den Wert einer Variablen ändern. Die dem zugrundeliegenden Operationen, die wir in der maschinennahen Programmierung noch selbst programmieren mussten, sind jetzt irrelevant geworden.

Diejenigen von Ihnen, die bereits in einer imperativen Sprache programmiert haben werden jetzt einwenden, dass Sie einer Variablen nicht nur einzelne Zahlen und Zeichen zuweisen können. Das ist streng genommen falsch. Was Sie meinen ist, dass wir einem Bezeichner oder einem Namen z.B. bei einer Funktion mehr als nur einzelne Zahlen oder Zeichen zuordnen können. Ein String beispielsweise ist aber bereits eine Datenstruktur und keine einfache Variable mehr. Doch bevor wir uns um diese Fälle kümmern können, müssen wir zunächst klären, was eine Datenstruktur ist, die Sie allerdings bitte nicht mit Datentypen verwechseln, die Sie zuvor kennen lernen.

Kontrolle

Variablen sind wie Register, nur mit dem Unterschied, dass Sie praktisch keine Beschränkung in der Anzahl Variablen haben, die Sie verwenden wollen. Außerdem steht es Ihnen weitgehend frei, Variablen so zu nennen, wie Sie das wollen. An dieser Stelle wurde der Begriff der Variablen sehr strikt definiert und beispielsweise nicht als Bezeichner von Datenstruktu-

ren verwendet.

Wichtig

Das ist nicht allgemeingültig, sondern es geht hier darum, dass Sie eine genaues Verständnis dafür bekommen, was eine Variable tatsächlich ist. Leider wird der Begriff für viele Dinge verwendet, selbst wenn diese Nutzung so nur für einzelne Programmiersprachen gilt. Wenn Sie dann eine andere Programmiersprache lernen, kommen Sie in Situationen, in denen Sie zum einen Variablen für bestimmte Dinge nicht nutzen können, bei denen Sie daran gewöhnt sind und zum anderen können Sie sie dann für etwas nutzen, was Ihnen vollkommen absurd vorkommt. Wenn Sie dagegen die strikte Definition des Begriffs Variable nutzen, den ich hier vorgestellt habe, wird Ihnen das deutlich seltener passieren.

10.6.4 Statisch versus nicht-statisch

In den einleitenden Kapiteln haben Sie ja bereits den Unterschied zwischen dynamisch und statisch typisierten Sprachen kennen gelernt. Aber es gibt eben nicht nur statische bzw. dynamische Datentypen, sondern noch wesentlich mehr, was bei einer Programmiersprache dynamisch sein kann. (Bei C ist das allerdings recht wenig.)

Dieser Unterschied zieht sich also quer durch die Programmierung und Sie werden dementsprechend immer wieder mit Sprachen zu tun haben, in denen Sie etwas programmieren, das entweder während des gesamten Programmablaufs gleichbleibt oder sich verändert bzw. sich verändern kann.

Leider gibt es hier jedoch keine einheitliche Nomenklatur. Das bedeutet, dass Sie nicht einfach im Handbuch zur Sprache nachschlagen können, welche Teile der Sprache nun statisch und welche dynamisch sind, bzw. wie Sie das jeweils festlegen können.

Bei Variablen ist es beispielsweise so, dass manchmal von konstanten Variablen und manchmal von statischen Variablen die Rede ist. Die Bezeichnung Konstante wie Sie sie aus der Mathematik kennen ist hier nicht zutreffend.

Erinnern Sie sich in diesem Bezug bitte daran, dass alles, was im Speicher eines Computers steht letztlich ein veränderlicher Wert ist: Es sind alles Daten, die jederzeit überschrieben werden können. Und der Quellcode, den Sie erstellen nutzt ja nur all das, was in einer Sprache bereits fest einprogrammiert ist. Ob diese Sprache (was bei C an vielen Stellen der Fall ist), dann einmalige Definitionen dauerhaft behält also statisch ist oder es

zulässt, dass diese im Verlauf eines Programms beliebig geändert werden also dynamisch ist, hängt eben von der Sprache selbst ab und nicht davon, was Sie erwarten.

Kontrolle

Während Sie bei der maschinennahen Programmierung lediglich Werte in Register laden, dort verändern und dann im Speicher ablegen konnten, müssen Sie bei allen Elementen einer höheren Programmiersprache lernen, ob diese während des Programmablaufs geändert werden können oder nicht. Der Unterschied wird mit unterschiedlichen Begriffen bezeichnet.

Synonym für statisch wird konstant verwendet.

Synonym für dynamisch wird nicht-statisch, variabel, veränderlich und eine Reihe weiterer Begriff verwendet.

Verwechseln Sie bitte nicht eine Variable als Bezeichner für einen Speicherbereich, worüber wir hier reden, mit einer Variablen als dem eigentlichen und/oder veränderlichen Wert. Der Begriff der Variable, den ich hier vorstelle beinhaltet sowohl einen Bezeichner als auch einen Datentyp als auch einen Wert, der an einer Speicherstelle gespeichert ist.

10.7 Deklaration, Initialisierung und der Scope

Wichtig: Wenn Sie eine Variable deklarieren, wird sie bei den meisten Programmiersprachen mit einem Standardwert initialisiert. Wenn Sie eine Sprache regelmäßig nutzen, sollten Sie diesen Standardwert kennen.

Beispiele für die Deklaration:

Die folgende Zeile deklariert eine Variable mit dem Bezeichner eineZahl als Variable vom Typ ganzzahlig mit 32 Bit. Der Bezeichner „eineZahl“ darf in diesem Programm vorher noch nicht aufgetaucht sein.

int eineZahl;

Die folgenden zwei Zeilen sollen nur deutlich machen, dass eine Deklaration nichts ist, worüber Sie groß nachdenken müssten:

char grosserBuchstabe; enum Farben;

Noch ein abschließender Hinweis: Das Semikolon zeigt dem Compiler an, dass hier eine Programmzeile zu Ende ist. Sie könnten also theoretisch auch diese Zeile nutzen, um alle drei Variablen zu deklarieren:

float pi; short bully; pointers zeiger;

Nur ist das eben schlecht lesbar. So programmieren nur Leute, die wollen, dass niemand Ihren Quellcode lesen kann (was aber nicht funktioniert) oder die nicht wissen, was sie tun.

Beispiele für die Initialisierung / Zuordnung eines Wertes:

Das folgende können Sie nur tun, wenn Sie die jeweiligen Variablen bereits initialisiert haben. Hier wird einer Variablen ein Wert zugeordnet:

eineZahl = 42;

Der Variablen eineZahl wird hier also der Wert 42 zugeordnet. Oben hatten wir eineZahl als int deklariert. Und da 42 eine ganze Zahl und innerhalb des Wertebereichs von int ist, können wir das tun.

Ist eine Variable erst einmal deklariert, können Sie ihr so oft neue Werte zuordnen, wie Sie wollen. Einsteigern fällt der Umgang mit Variablen deshalb manchmal etwas schwer, aber machen Sie sich da keine Gedanken, das wird Ihnen innerhalb kurzer Zeit in Fleisch und Blut übergehen; Sie müssen nur das tun, was fürs Lernen einer Programmiersprache das wichtigste ist: Programmieren, Programmieren und nochmal Programmieren.

Sie können bei C und vielen anderen Programmiersprachen auch eine Variable deklarieren und initialisieren. Das sähe dann so aus:

int nochEineZahl = 937;

In den Fällen, in denen Sie darauf angewiesen sind, mit Fließkommazahlen zu arbeiten, müssen Sie beachten, dass hier die anglo-amerikanische Notation für die Dezimaltrennung gilt. Umgangssprachlich ausgedrückt: Wo Sie bei Zahlen ein Komma verwenden, müssen Sie einen Punkt setzen! Hier das entsprechende Beispiel:

pi = 3.1415;

Der Scope und das Ende von Variablen

Wie Sie jetzt wissen, erzeugen Sie Variablen entweder durch eine Operation oder durch eine Deklaration. Und irgendwo müssen all die Variablen im Rechner verwaltet werden, damit sie nicht einfach überschrieben werden. Das ist Ihnen wahrscheinlich gar nicht klar: Sie wissen, dass Sie Variablen erzeugen, ihnen Werte zuordnen und diese ändern können. Aber über die Verwaltung des Speichers haben wir gar nicht gesprochen, denn das macht ja „die Programmiersprache".

Folgendes Beispiel soll Ihnen das Problem vor Augen führen, das Ihnen bislang entgangen ist: Stellen Sie sich vor, Sie würden ein Programm verfassen. Alle Variablen, die dieses Programm erzeugt würden im Speicher erhalten bleiben. Zusätzlich arbeiten ja noch viele andere kleine und große Programme (als Teil des Betriebssystems) im Rechner, die auch massenhaft Variablen erzeugen. Und nehmen wir an, all diese Variablen würden ebenfalls im Speicher verbleiben. Was würde dann wohl innerhalb kürzester Zeit passieren?

...

Genau! Der Speicher wäre irgendwann voll und der Rechner würde gar nichts mehr tun oder damit beginnen, bereits durch alte Variablen belegten Speicher mit neuen Werten von anderen Variablen zu füllen. Kurz gesagt: Chaos.

Ein Teil der Lösung, um solches Chaos zu vermeiden nennt sich Garbage Collector, aber damit haben Sie bei der Programmierung nichts zu tun, denn es ist ein Teil der Laufzeitumgebung, der kontinuierlich prüft, welche Speicherbereiche von keinem Teil des Programms mehr genutzt werden. Diese Speicherbereiche gibt er dann wieder frei, sodass sie für andere Aufgaben verwendet werden können. Ein anderer Teil wird als Scope oder Rahmen bezeichnet und gehört zu den Dingen, die Sie fest einprogrammieren. Der Scope ist jeweils ein Bereich, in dem Bezeichner von Variablen gültig sind. Das selbe gilt für die Bezeichner von anderen Dingen wie Funktionen, über die wir noch nicht gesprochen haben.

Bei C wird der Scope durch geschweifte Klammern abgegrenzt.

Beispiel:

Innerhalb eines C Programms sehen Sie folgende Zeilen:

```
int main(void)
{
int zahl1 = 1;
int zahl2 = 2;
return zahl1 + zahl2;
}

float addiere(float a, float b)
{
float zahl1 = a;
float zahl2 = b;
return zahl1 + zahl2;
}
```

Sie wissen jetzt zwar nicht, was Sie mit der Zeile float addiere(float a, float b) anfangen sollen, aber das spielt hier keine Rolle. Denn hier reden wir nur über den Scope und für den spielt das keine Rolle.

Da Sie jetzt wissen, was der Scope ist, wissen Sie auch, dass in diesem Programmfragment zweimal eine Variable vom Typ int deklariert wird, die mit dem Bezeichner zahl1 angesprochen wird. Sie wissen jetzt auch, dass beide Variablen nichts miteinander gemein haben, weil Sie jeweils in unterschiedlichen Scopes enthalten sind.

Nun lassen sich geschweifte Klammern jeweils beliebig verschachteln und wie immer gilt, dass bei jeder Programmiersprache individuell geregelt ist, ob eine Variable in den inneren Rümpfen eines Scope verwendet werden kann oder nicht.

Es ist ein guter Programmierstil jeweils am Anfang eines Rumpfes die Deklaration aller (nicht-anonynmen) Variablen durchzuführen, die innerhalb dieses Rumpfes verwendet werden, da man so doppelte Verwendungen ausschließt.

Kontrolle

So wie Sie bei der maschinennahen Programmierung mit Zahlen arbeiten, die in sogenannten Registern verarbeitet werden, arbeiten Sie bei der imperativen Programmierung mit sogenannten Variablen, die Sie deklarieren und initialisieren müssen.

Sie wissen, dass es im Gegensatz zu Speicherbereichen bei der maschinennahen Programmierung Standardwerte je nach Datentyp gibt, mit denen deklarierte Variablen automatisch initialisiert werden, und dass Sie diese

Standardwerte kennen sollten, wenn Sie längere Zeit mit einer Sprache programmieren.

Innerhalb eines Scope darf bei C ein Bezeichner vor der Deklaration nicht auftauchen. Warum?

Übung

Damit Sie jetzt ein wenig Programmierpraxis bekommen, hier eine kleine Übung: Starten Sie eine IDE oder einen Editor Ihrer Wahl, um ein C-Programm zu entwickeln.

(1) Nehmen Sie nun den Quelltext vom Anfang dieses Unterkapitels. Unterhalb dieses Absatzes finden Sie nun eine Fassung, in der eine Variable deklariert, initialisiert und ausgegeben wird.

Lassen Sie sich hier bitte nicht davon irritieren, dass nach dem Rumpf kein Semikolon auftaucht: Der Rumpf/Scope legt fest, in welchem Bereich Bezeichner definiert sind. Das Semikolon dagegen legt fest, wo ein Befehl endet.

Dann wären da noch die Einrückungen anzusprechen. Diese dienen in C der Lesbarkeit: Eine allgemein übliche Konvention lautet, dass die Zeilen innerhalb eines Rumpfes um drei Leerstellen eingerückt werden. Gerade wenn Sie mit gestaffelten Rümpfen (also Rümpfe in Rümpfen in Rümpfen in ...) arbeiten, werden Sie das bei Änderungen zu schätzen wissen.

Beachten Sie bitte, dass es Programmiersprachen gibt, bei denen ein Scope ausschließlich durch solche Einrückungen festgelegt wird.

Der folgende Quellcode ist auch für IDEs geeignet, da er einige Ergänzungen enthält, die Sie jedoch erst dann verstehen können, wenn Sie den Abschnitt über Funktionen gelesen haben. Deshalb empfehle ich Ihnen, weiterhin mit einem Editor und dem Quellcode zu arbeiten, den Sie am Beginn des Kapitels finden können.

```
#include <stdio.h>
int main(void) {
int eineZahl = 42;
printf("Der Wert der Variablen ist: %d" , eineZahl);
return 1;
}
```

Das folgende gilt vorrangig für diejenigen von Ihnen, die eine IDE nutzen:

Auch wenn Sie das inzwischen wissen sollten, hier nochmal der Ablauf zum funktionierenden Programm: Nachdem Sie die Datei bzw. das Projekt gespeichert haben, müssen Sie es kompilieren und linken (dafür gibt es in IDEs eine Schaltfläche oder einen entsprechenden Menüeintrag). Zusammengefasst werden kompilieren und linken auch als build bezeichnet. Wenn Sie alles richtig abgetippt haben und die IDE richtig konfiguriert ist, erscheinen einige Zeilen in einem Fenster der IDE, die im Grunde nur besagen, dass kompilieren und linken erfolgreich verlaufen sind. Anschließend starten Sie das Programm.

(2) Auch wenn es nicht allzu spektakulär ist, folgt eine erweiterte Fassung des Quellcodes.

```
#include <stdio.h>
int main(void) {
int eineZahl = 42;
printf("Der Wert der Variablen ist: %d\n" , eineZahl);
char buchstab = ´a´;
printf("Der Buchstabe lautet: %c" , buchstabe);
return 1;
}
```

Sehen Sie sich an, welche Unterschiede es bei der Ausgabe gegenüber dem ersten Quelltext gibt und stellen Sie Vermutungen an, was diese Änderungen bewirken. Versuchen Sie dann basierend auf Ihren Annahmen das Programm erneut zu erweitern, indem Sie eine Variable vom Typ float mit dem Wert 15.293 hinzufügen und diese in einer weiteren Zeile ausgeben lassen.

Wenn Sie alles richtig gemacht haben, werden die folgenden Zeilen ausgegeben:

Der Wert der Variablen ist: 42 Der Buchstabe lautet: a Die Fließkommazahl ist: 15.293000

(3) Versuchen Sie jetzt durch eine Internetrecherche herauszufinden, was Sie ändern müssen, damit in der letzten Zeile nicht 15.293000 sondern einfach nur 15.293 steht.

(4 - Schwierig) Und jetzt ändern sie das Programm so ab, dass in der letzten Zeile 15,293 steht. Wichtig: Die Aufgabe ist nur dann richtig gelöst, wenn Sie anschließend den Wert der Variablen von 15.293 auf beliebige andere

Fließkommazahl ändern können und das Programm diese anderen Zahlen mit dem Komma anstelle des Punktes ausgibt. Für die Lösung ist es akzeptabel, wenn das Programm eine feste Anzahl an Stellen nach dem Komma ausgibt. Die Nachkommastellen brauchen hier also nicht vollständig oder richtig gerundet sein.

Wichtiger Hinweis

Wenn Sie mit dem Quellcode da oben ein wenig herumspielen, werden Sie feststellen, dass zwar viele Stellen genauso eingetippt werden müssen, wie das hier zu lesen ist. Aber andere Dinge können Sie relativ beliebig ändern und trotzdem gibt das Programm dieselbe Ausgabe aus.

Wenn Sie beispielsweise Ihr Programm so ändern, dass Sie alle Deklarationen an den Anfang des Rumpfes setzen, dann wird sich am Ablauf nichts ändern. Erinnern Sie sich noch, warum Sie das tun sollten?

10.8 Operationen und Funktionen

Bislang haben Sie des Öfteren davon gelesen, dass hier von Befehlen die Rede war. Wenn Sie mit Programmierern reden, dann werden Sie diese Bezeichnung dort eher selten hören. Vielmehr gibt es hier drei Begriffe, die Sie als Synonym für einen Befehl verstehen könnten, die Sie aber klar unterschieden müssen. In der imperativen Programmierung kommen zwei davon zum Einsatz: Operationen und Funktionen. Bei der objektorientierten Programmierung gibt es dann noch die sogenannten Methoden, die sich aber in der Programmierung nur durch ein Detail von Funktionen unterscheiden, das so trivial ist, dass es Ihnen wahrscheinlich nicht einmal auffallen wird, wenn es Ihnen niemand erklärt. Aber bleiben wir vorerst bei der imperativen Programmierung.

Operationen kennen Sie aus der Mathematik: Dort haben Sie die sogenannten Algebren kennen gelernt. Da aber die meisten Studienanfänger diese Lektion bereits wieder vergessen haben, folgt hier eine kleine Einführung... Alpers berüchtigte Algebra BROT: (In der Mathematik wird der Begriff einer Algebra anders verstanden; das hier ist ein Beispiel dafür, wie InformatikerInnen sich ein gutes Konzept der Mathematik ausleihen, um es für Ihre Zwecke zu nutzen... Also das, was alle Wissenschaften der Mathematik antun.)

10.8.1 Die Algebra BROT

Sie werden sich erinnern: In den einleitenden Kapiteln haben Sie etwas Verwirrendes gelesen: Demnach hat Mathematik eigentlich nur am Rande etwas mit Zahlen zu tun und vielmehr ginge es darum, neue Welten und Universen zu beschreiben. In den nächsten Absätzen werde ich Ihnen ein Beispiel dafür geben.

Zunächst beginnen einige InformatikerInnen seit einigen Jahrzehnten eine Argumentation damit, dass sie zunächst eine Menge definieren. Eine solche Menge definieren sie über die Eigenschaften, die alle Objekte haben, die Teil dieser Menge sind. Es geht dabei aber nicht darum, dass die Ausprägung der Eigenschaften beschrieben wird, sondern dass definiert wird, um welche Eigenschaften es sich handelt. Definitionen der verschiedenen Mengen von Zahlen haben Sie schon so oft gesehen, dass eine weitere Wiederholung unsinnig wäre. Also nehmen wir die Menge BROT. Ja, richtig gelesen, hier gibt's keine Zahlen, hier gibt's nur Brote. Und ja, das ist Mathematik oder zumindest das, was wir als InformatikerInnen von der Mathematik übrig lassen.

Trommelwirbel... und ein kurze Pause, damit Sie Ihrem Gehirn die Gelegenheit zur Beruhigung geben können.

Unsere Menge heißt also BROT. Und was wird darin sein? Natürlich jede Menge Brote. Und wie definieren wir so etwas? Genau, wir notieren, welche Eigenschaften ein Brot haben kann.

Boshafte Naturen hätten diese Menge so definiert, dass darin alle möglichen Dinge gesammelt werden würden, die nicht das Geringste mit Brot zu tun haben, denn verboten ist das nicht. Im realen Leben hören Sie ja auch pausenlos Bezeichnungen, die nicht dem naiven Eindruck entsprechen:

• Nehmen wir eine Bezeichnung wie Gleichstellungsbeauftragte. Hier würden Sie naiv an einen Menschen (unabhängig vom Geschlecht) denken, dessen Aufgabe darin besteht Benachteiligungen auszuhebeln, die aufgrund des Geschlechts, der religiösen Zugehörigkeit oder anderer Aspekte bestehen, die aber nicht in der Leistung bzw. Leistungsfähigkeit der/des Einzelnen bestehen. Tatsächlich verbirgt sich hinter dieser Bezeichnung gelegentlich eine Frau, die ausschließlich die Belange von Frauen für Frauen vertritt. Dabei gehört es zur Allgemeinbildung, dass beispielsweise im erzieherischen Bereiche Männer aufgrund sexistischer Klischees häufig benachteiligt oder diffamiert werden.

• Oder denken Sie an die sogenannten sozialen Netzwerke. Ein soziales

Netzwerks besteht aus Menschen, die miteinander in Kotakt stehen. Webplattformen dagegen, die als soziale Netzwerke bezeichnet werden sind in aller Regel Sammelwerke, deren einziger Zweck darin besteht, ein allumfassendes Profil über die hier registrierten Nutzer zu erstellen. Dabei bedeutet allumfassend, dass selbst der Psychotherapeut eines Nutzers einer solchen Webplattform nach jahrelanger Therapie keinen derart tief gehenden Einblick in die Persönlichkeit und Lebensumstände des/der Betreffenden gewonnen haben kann.

Aber wieder zurück zum Thema.

Aufgabe:

Sammeln Sie einige Eigenschaften die bei verschiedenen Arten von Brot vorkommen. Wenn Ihnen nicht klar ist, was damit gemeint ist, dann denken Sie an eine Menge von Autos. Hier wäre Farbe eine mögliche Eigenschaft, denn abgesehen von einigen Prototypen hat im Regelfall jedes Auto wenigstens eine Farbe. Also nochmal: Was für Eigenschaften hat Brot? (Machen Sie sich hier keine Gedanken, wenn Sie eine Eigenschaft nicht in einem Wort zusammenfassen können, formulieren Sie die Eigenschaften ruhig zunächst als Sätze.)

Hier sind wir auch beim ersten Beispiel, was den Informatiker vom Programmierer unterscheidet: Der Programmierer geht jede Aufgabe konkret an und entwickelt für das konkrete Problem eine konkrete Lösung. Im Gegensatz dazu betrachtet ein Informatiker zunächst, wie das Problem auf abstrakter Ebene aussieht und nutzt dann ähnlich abstrakte Lösungsansätze, um einen Lösungsweg zu entwickeln, den er dann z.B. in Form eines Computerprogramms konkretisiert. Dieser Abstraktionsschritt wird auch als Modellieren bezeichnet, eine Methodik, die alle Naturwissenschaftler nutzen.

Wenn wir uns jetzt unsere Menge BROT ansehen, so lassen sich leicht einige Eigenschaften finden. Ein Brot kann saftig oder krümelig sein. Es hat eine Farbe, die häufig zwischen weiß und dunkelbraun schwankt, aber je nach Zusatzstoffen oder Zustand auch rötlich oder grünlich sein kann. Es hat eine Form irgendwo zwischen Fladen und Kasten. Es kann aus reinem Mehl bestehen, aus Mehl und verschiedenen Körnern, usw. Wenn Sie weiter überlegen, dann werden Ihnen hier noch viele Eigenschaften einfallen.

Sie könnten nun Variablen deklarieren, die jeweils einer Eigenschaft unserer Menge BROT entsprechen. Hier haben Sie auch schon ein Beispiel dafür, wie Mathematik und Programmieren sich in der Informatik treffen. Richtig spannend wird dieser Aspekt aber erst in der objektorientierten Programmierung.

Von der Menge zur Algebra

Das folgende ist keine präzise mathematische Einführung. Wenn Sie hier eine genaue Erklärung haben möchten, dann sprechen Sie bitte einen Mathematiker an.

Den Begriff Algebra haben Sie schon in der Schulzeit immer wieder gehört, aber wahrscheinlich nie eine Erklärung dazu gehört, also eine Antwort auf die Frage, was eine universelle Algebra oder eine algebraische Struktur ist. Eine algebraische Struktur besteht zum einen aus einer Menge (damit ist das gemeint, was wir soeben als Menge erarbeitet haben) und zum anderen aus einer beliebigen Anzahl Operationen. Das was eine algebraische Struktur dabei auszeichnet ist, dass jede dieser Operationen mit jedem Element der Menge der algebraischen Struktur nutzbar ist. Ein einfaches Beispiel für die Operation wäre die Addition und ein einfaches Beispiel für die Menge wären die natürlichen Zahlen: Egal auf welche zwei natürlichen Zahlen Sie eine Addition anwenden, das Ergebnis ist ebenfalls eine natürliche Zahl.

Operationen sind umgangssprachlich Tätigkeiten, es geht also bei der Definition einer algebraischen Struktur darum, eine Ansammlung von Objekten abzugrenzen, mit der bestimmte gleichartige Dinge getan werden können. Die Mathematik besteht dann darin, zu prüfen, welche weitergehenden Möglichkeiten sich aus den definierten Eigenschaften und Operationen ergeben und ob wir dabei spannende und elegante Abkürzungen finden können.

Das allerdings ist bei der Programmierung zunächst nicht relevant. Also zurück zu den Operationen. Lassen Sie uns nun zur Algebra BROT kommen, nachdem wir gerade die Menge BROT definiert haben. Richtig: Wir benutzen hier einen Bezeichner, namentlich BROT, um zwei unterschiedliche Dinge zu bezeichnen. Dürfen wir das? Klar, in der Mathematik dürfen wir das schon, in der Programmierung dagegen nicht. Der Grund ist einfach: Die Mathematik setzt den gesunden Menschenverstand voraus, also die Fähigkeit, beispielsweise zu erkennen, in welchem Umfeld ein Problem angesiedelt ist und daraus auf eine sinnvolle Lösung zu schließen. Die Programmierung dagegen… Nun Sie können diesen Satz in eigenen Worten beenden und können damit auch gleich auf die Antwort zur Frage schlussfolgern, warum ProgrammiererInnen nicht das gleiche wie InformatikerInnen sind.

Um also von der Menge BROT zur algebraischen Struktur BROT zu kommen, müssen wir jetzt also einige Tätigkeiten oder Anwendungen benennen, die mit einem Element der Menge Brot durchgeführt werden kann.

Dabei spielt es keine Rolle, ob und wenn ja welchen Sinn die Anwendung dieser Operation macht. Es geht nur darum, ob sie mit jedem Element möglich ist.

Und hier nehmen wir den klassischen Wortwitz: Kann man Brot einfrieren? Na sicher kann man, Mann, Frau, Zombie, Echse, Spock oder wer auch immer das tun. Also haben wir unsere erste Operation: Einfrieren. Wobei so etwas eine 1-stellige Operation ist, was recht langweilig ist. (Die Stelligkeit einer Operation besagt etwas darüber, wie viele Elemente wir benötigen, um die Operation durchzuführen.)

Wie gesagt: Es ist nicht wichtig, ob diese Operation Sinn macht; wichtig ist ausschließlich, dass sie auf jedem Element der Menge (hier der Menge BROT) definiert ist. Die Aussage „Die Operation X ist definiert für Y." bedeutet hier nichts anderes, als dass es möglich ist, die Operation X mit dem Element Y durchzuführen. Wenn Y eine Menge ist, dann bedeutet diese Aussage, dass die Operation X ausnahmslos auf jedem Element der Menge Y ausgeführt werden kann. Sonst wäre nämlich unsere Kombination aus Menge und Operationen keine algebraische Struktur mehr.

Wenn Sie wie bei der Definition der Menge BROT jetzt die auf dieser Menge definierten Operationen notieren und beides gemeinsam aufschreiben, dann haben Sie schon die Algebra BROT definiert. Das folgende wäre also eine mögliche Definition der Algebra BROT, wobei hier P die Struktur bzw. das Modell ist und F die Operationen zusammenfassen:

BROT:

P = (Knusprigkeit — Farbe — Temperatur — Form) F = (einfrieren — schneiden — bestreichen — essen — backen — stapeln)

Anmerkung: Formal werden Algebren ganz anders notiert, aber dazu sprechen Sie bitte den Mathematikdozenten Ihres Vertrauens an.

Von der algebraischen Struktur zur programmierten Operation

Sie fragen sich jetzt vielleicht, wie die Operationen ausgeführt werden können, denn darüber haben wir hier noch nicht gesprochen. Aber weil dies keine Einführung in die Mathematik ist, werden wir darauf auch nicht weiter eingehen. Vielmehr sollte dieser Teil dazu dienen, dass Sie eine Vorstellung davon bekommen, warum die Mathematik (bei diesem Beispiel speziell die Algebra) eine so wichtige Rolle in der Informatik übernimmt. Aber wir kommen jetzt wieder zur Programmierung zurück, wo wir uns ansehen werden, was Operationen bei einem Programm sind.

Operationen sind hier so etwas wie Aktivitäten, die für jeden Datentypen einzeln definiert sind. Um das mathematische Modell von eben zu nehmen: Stellen Sie sich jeden Datentyp als eine algebraische Struktur vor. Dann sind Sie auch direkt beim Verständnis davon, was nun eine Operation ist: Es ist das, was Sie mit jeder Variablen dieses Datentyps tun können. Operationen werden dabei gewissermaßen mit der Programmiersprache ausgeliefert, Sie können Sie also nicht einprogrammieren, sondern nutzen Sie als einfachste Befehlsformen.

Kontrolle

Datentypen und Operationen verhalten sich zueinander wie Mengen und Operationen in der Mathematik. Operationen stellen einfachste Befehle dar, mittels derer Sie Variablen ändern können.

Genau wie Datentypen sind Operationen in C statisch.

Aufgabe:

Überlegen Sie sich, was es bedeuten würde, wenn Operationen dynamisch wären. Vergessen Sie an dieser Stelle bitte nicht, dass es hier nicht um die Frage geht, ob das Sinn macht, sodern nur um die Frage, welche Folgen das hätte. Und da gibt es vieles, was Ihnen einfallen könnte.

Mathematische Operationen

Mathematische Operationen sind für ganzzahlige Datentypen und für Datentypen mit Fließkommazahlen definiert. Wichtig ist aber, dass Sie bei jeder Programmiersprache lernen, wie sich die Operationen genau verhalten. Bevor wir dazu kommen, hier die Übersicht: (a, b sind beliebige Variablen)

• Addition: a + b • Subtraktion: a – b • Multiplikation: a * b • Division: a / b • Modulo-Rest: a % b

Wichtig: Eine Operation wie a – b reduziert nicht (!) a um den Wert von b. Deshalb müssen Sie das Ergebnis einer Operation in aller Regel einer Variablen zuordnen.

Wiederholung: Wenn Sie tatsächlich die Variable a um den Wert der Variablen b reduzieren wollen, dann können Sie das tun, indem Sie das Ergebnis der Operation a – b der Variablen a zuordnen. Im Gegensatz zu dem, was

Sie im Mathematikunterricht gelernt haben, macht also die folgende Programmzeile tatsächlich Sinn. (Wenn Sie das nicht verstehen, arbeiten Sie bitte nochmal den Abschnitt zu anonymen Variablen durch.)

a = a – b

Aufgabe: Begründen Sie mit Ihrem bisherigen Wissen aus diesem Kurs, warum die Operation a – b den Wert von a nicht reduziert.

Wichtig: Beachten Sie jedoch, dass jede Operation bei statisch typisierten Sprachen im Regelfall nur dann definiert ist, wenn beide Elemente vom gleichen Datentyp sind. Es kann Ihnen also passieren, dass Sie eine Operation, die für zwei unterschiedliche Datentypen definiert ist, dennoch nur mit Variablen des selben Datentyps durchführen können. Das ist für jede Programmiersprache individuell festgelegt; Sie können es nicht durch logische Schlussfolgerung feststellen, sondern müssen die Dokumentation der jeweiligen Sprache zu Rate ziehen.

Aufgabe 1: Prüfen Sie das nach, indem Sie die je zwei ganzzahlige und zwei Fließkomma-Variablen deklarieren und initialisieren. Dann programmieren Sie alle mathematischen Operationen, wenn beide Operanden vom gleichen Typ sind. Dann programmieren Sie sie, wenn der erste Operand ganzzahlig und der zweite Operand ein Fließkommawert ist. Abschließend umgekehrt. Sie müssten also zwanzig Zeilen Code mit der Berechnung und Ausgabe eines Ergebnisses programmieren. Sie brauchen das Ergebnis der Operationen keiner neuen Variablen zuzuordnen, sondern können die Operation anstelle einer Variablen in die Klammern von printf() eintragen.

(gehört zur Aufgabe) ACHTUNG: Dieser Code wird eine Vielzahl von Fehlermeldungen erzeugen und die Hauptaufgabe für Sie besteht darin, diese Fehlermeldungen zu lesen und sich zu überlegen, was zu dem Fehler geführt hat. Das können Sie aber nur dann, wenn Sie klar zwischen den einzelnen Datentypen abgrenzen und die Fehlermeldungen konzentriert lesen.

Und ja, so etwas kann Teil einer Prüfungsaufgabe sein.

Aufgabe 2: Überlegen Sie sich bei jedem Fehlerfall, was Sie im Quellcode ändern müssten, damit Sie eine Lösung bekommen, die dem entspricht, was Sie erwarten würden.

Aufgabe 3: Rechnen Sie die Ergebnisse nach (auch und gerade bei den Operationen, die keine Fehlermeldung erzeugt haben) und überlegen Sie

sich bei den entsprechenden Fällen, warum das Ergebnis nicht mit dem übereinstimmt, was Sie jeweils erwarten würden. (Wenn Sie jeweils das Ergebnis erwartet haben, das der Rechner ausgegeben hat, dann notieren Sie, warum es nicht mit dem Ergebnis übereinstimmt, das Sie erhalten hätten, wenn Sie die arithmetischen Regeln aus dem Mathematikunterricht angewendet hätten.)

10.9 Funktionen

Den Begriff einer Funktion kennen Sie aus dem Mathematikunterricht. Naiv könnte man ihn so beschreiben: Eine Funktion wäre dann eine Sammlung von Operationen, die aus einem oder mehreren Elementen einer Menge ein Element einer anderen oder der selben Menge erzeugt. So können Sie beispielsweise eine Funktion auf einen Vektor anwenden, der die Länge des Vektors berechnet. Der Vektor ist Element einer Menge, die Länge des Vektors Teil einer anderen Menge.

Eine andere Erklärung einer Funktion ist die Abbildung eines Wertes einer Menge auf ein Element einer (ggf. anderen) Menge.

Die Mathematik beschäftigt sich nun weiter mit den Eigenschaften von Funktionen, Ähnlichkeiten und Unterschieden zwischen Gruppen von Funktionen und was sich daraus an weitergehenden Aussagen über diese Gruppen von Funktionen ableiten lässt. Also wenn Sie mich fragen, ist das wirklich spannend, gerade wenn man es auf komplexe reale Probleme anwendet, aber Sie wollen ja nur lernen wie man programmiert. Also schauen wir wieder auf den Funktionsbegriff in der Programmierung.

Und dort ist eine Funktion wieder etwas, das in aller Regel einen Bezeichner trägt (wobei auch sogenannte anonyme Funktionen möglich sind, aber die interessieren uns für den Moment nicht weiter) und dem Sie eine beliebige Anzahl an Variablen übergeben können. Wie viele das sind, wird bei der Programmierung einer Funktion festgelegt. Sie werden im Rahmen einer Funktion als Argumente bezeichnet, um sie sprachlich eindeutig von den Variablen eines Programms unterscheiden zu können. Denn die Funktion verarbeitet Kopien der Werte der Variablen (ohne den Wert der Variablen selbst zu ändern) und gibt einen eigenständigen Wert zurück. Und diese Kopien sind eigenständige Variablen, die als Argumente bezeichnet werden.

In C gilt, dass Funktionen statisch sind.

Wichtig: Auch wenn der Rumpf einer Variablen ein eigenständiger Scope

ist, sollten Sie Variablenbezeichnungen, die Sie außerhalb einer Funktion eingeführt haben nicht innerhalb der Funktion verwenden, selbst wenn Sie mit einer Sprache arbeiten, die Ihnen das ermöglicht. Die Lesbarkeit und damit die Möglichkeit Ihr Programm später zu verbessern wird dadurch deutlich verschlechtert.

Es gibt noch Sonderformen von Funktionen: Funktionen benötigen nicht unbedingt Argumente, um ihre Aufgabe zu erfüllen. Und es ist möglich Funktionen zu programmieren, die keinen Rückgabewert erzeugen. Für die drei Varianten gibt es keine eigenständigen Bezeichnungen, wir sprechen unabhängig von der Variante grundsätzlich nur von einer „Funktion“. Und schließlich ist es noch möglich, Funktionen zu programmieren, die weder Argumente noch einen Rückgabewert haben. Am einfachsten ist das nachvollziehbar, wenn Sie an die main() denken. Aber andere Fälle sind ebenfalls denkbar. Im Alltag werden Sie hier beim Programmieren auch gar nicht differenzieren und wie alle Programmierer immer „nur“ von einer Funktion reden.

Sehen wir uns die drei ersten Fälle im Detail anhand von Beispielen an: Variante a) Sie möchten eine Funktion haben, die Ihnen den größten gemeinsamen Teiler zweier Zahlen ausgibt. Ohne auf die Programmierung der Funktion einzugehen, können wir also sagen, dass wir eine Funktion haben, die zwei Argumente benötigt und einen Rückgabewert hat. Alle müssen vom Datentyp her ganzzahlig sein. In allgemeiner Form könnten Sie das dann so deklarieren:

```
int ggT (int argumente1, int argument2){ ... }
```

Diese Form ist standardisiert, Sie haben hier also nur bei der Bezeichnung der Argumente, bei der Wahl des jeweiligen Datentyps und beim Namen der Funktion gewisse Freiheiten. Um hier Missverständnisse zu vermeiden sei gesagt, dass in diesem Beispiel die Funktion nur deshalb genau zwei Argumente erhält, weil sie ja den größten gemeinsamen Teiler zweier Zahlen berechnen soll. In anderen Worten: Die Anzahl Argumente einer Funktion hängt davon ab, wie viele Argumente Sie verwenden wollen. Es gibt weder Vorschriften noch logische Gründe, die eine bestimmte Anzahl Argumente in irgend einer Form festlegen.

Die Deklaration da oben ist wie folgt zu lesen:

• Das erste int gibt den Datentyp des Rückgabewerts an. Wir können hier problemlos int nehmen, wenn dieser Datentyp für die Argumente ausreicht, denn der ggT zweier Zahlen kann ja nie größer sein, als die Zahlen selbst. In Sprachen wie C kann eine Funktion nur genau einen Rückgabewert

haben. Allerdings sind Sie hier nicht auf Variablen beschränkt, sondern können durchaus Datenstrukturen zurückgeben lassen. (Keine Sorge, Datenstrukturen haben wir noch nicht behandelt, aber sie folgen in Kürze.) • Dann folgt die Bezeichnung bzw. der Name der Funktion. Hier haben wir die Abkürzung ggT für größter gemeinsamer Teiler gewählt. Aber die Bezeichnung steht uns frei. Wir hätten diese Funktion auch funktionAlpha927 nennen können; für die Programmiersprache macht das keinen Unterschied. • Nach dem Funktionsnamen folgt stets ein Klammernpaar, in dem die zu übergebenden Argumente jeweils mit dem benötigten Datentyp stehen. Wichtig: Diese Bezeichnungen sollten im Programm noch nicht verwendet worden sein. Denn sonst kann es passieren, dass die Funktion falsche Ergebnisse liefert. • Abschließend folgt der Rumpf der Funktion. In diesem stehen mehrere Programmzeilen, von denen die letzte mit dem „Befehl" return beginnt. Hinter dem return muss eine Variable stehen, deren Datentyp derselbe ist, wie derjenige, den Sie für den Rückgabewert der Funktion definiert haben. Wie gewohnt gilt hier: Ob die Funktion das tut, was Sie wollen, hängt davon ab, ob Sie sie so programmiert haben oder nicht. Wenn Sie also den Funktionsrumpf so gefüllt haben, dass die Funktion etwas ganz anderes zurückgibt, dann beklagen Sie sich nicht, sondern konzentrieren Sie sich, um Ihre Denkfehler zu finden. Und hier auch ein ganz wichtiger Praxistipp: Wenn Sie schon seit Stunden programmieren, bewirkt eine Pause oft wahre Wunder. Oft hilft auch der Blick eines Nachbarn weiter.

Variante b) Die Funktion benötigt keinen Eingabewert. Hier ist das Klammernpaar hinter dem Funktionsnamen schlicht leer. Der Rest ist wie gehabt. Sie fragen sich, was solche eine Funktion soll? Nehmen wir an, Sie möchten eine Funktion haben, die Ihnen einen bestimmten Satz an Informationen über den Status Ihres Programms gibt. Dann brauchen Sie der Funktion beim Aufruf nicht mitzuteilen, welche Speicherstellen oder Variableninhalte Sie erhalten wollen, da es immer dieselben sind.

Variante c) Die Funktion hat keinen Rückgabewert. Dann dürfen Sie jedoch nicht einfach mit dem Namen der Funktion beginnen, sondern Sie benutzen den „Datentyp" void. Eine solche Funktion haben Sie schon kennen gelernt: printf() benötigt zwar immer ein Argument, aber da dieses Argument dann auf dem Bildschirm ausgegeben werden soll und Sie keine Variablen ändern wollen, macht hier ein Rückgabewert nur selten Sinn.

Kontrolle

Im Quellcode, mit dem Sie bislang gearbeitet haben, kommen zwei Funktionen vor. Wie heißen diese und was tun Sie?

Ansonsten sollten Sie wissen, wie eine Funktion deklariert wird, was der Rumpf einer Funktion ist und welche Bedeutung das Schlüsselwort return hat. (Überlegen Sie beispielsweise, was mit Programmcode passiert, der innerhalb eines Funktionsrumpfes in der Zeile nach einem return steht. Und welche Bedingung muss der Rest der Zeile erfüllen, die mit return beginnt?)

Wenn etwas ein Schlüsselwort ist, dann bedeutet das schlicht, dass dieser Bezeichner eine feste Bedeutung in der einer Programmiersprache hat. Sie dürfen Schlüsselwörter also nur für genau die Aufgabe verwenden, die in der jeweiligen Sprache vorgesehen ist. Das Schlüsselwort return ist hier nur ein Beispiel. In aller Regel gibt es für jede Sprache eine Übersicht der Schlüsselwörter und meist ist diese Liste auch nicht allzu lang.

Aufgabe

Die folgende Aufgabe dient wieder vorrangig dazu, dass Sie lernen, mit den Fehlerausgaben des Compilers zurecht zu kommen. Denn Sie werden beim Programmieren sehr oft kleine Fehler machen, die erst beim Kompilieren erkennbar werden. Dann ist es wichtig, dass Sie mit den Fehlermeldungen des Compilers etwas anfangen können. Also über wir genau das.

Hinweis: In der Aufgabenstellung ist ein simpler Fehler eingebaut, der Sie etwas verwirren könnte, wenn Sie die Erklärungen zu statisch und dynamisch typisierten Sprachen aufmerksam gelesen haben. Programmieren Sie die Aufgabe dennoch bitte in der geschilderten Reihenfolge und führen Sie dann die folgenden Punkte aus:

• Nachdem Sie das Programm fertig gestellt und kompiliert haben, kopieren Sie sämtliche Fehlermeldungen in ein Textdokument. • Beschreiben Sie in eigenen Worten, was die erste Fehlermeldung bedeuten könnte. • Schreiben Sie dann das auf, was Sie am Quellcode ändern wollen, um diese Fehlermeldung zu bereinigen. • Machen Sie so lange mit dieser Übung weiter, bis der Quellcode erfolgreich kompiliert wird und die gewünschte Ausgabe erfolgt. Hier das Programm, das sie erstellen und dann korrigieren sollen: • Deklarieren und initialisieren Sie vier Zahlen in einem C-Programm, denen Sie die Buchstaben a bis d zuordnen. • Deklarieren Sie dann eine Fließkomma-Variable mit dem Namen ergebnis1, der Sie die Funktion rechne(a, b, c, d) zuordnen. • Deklarieren Sie dann eine Fließkomma-Variable mit dem Namen ergebnis2, der Sie die Funktion rechne(b, a, c, d) zuordnen. • Programmieren Sie dann eine Ausgabe, in der die Werte von ergebnis1 und ergebnis2 auf den Bildschirm ausgegeben wird. (Sie können hier beliebig zusätzlichen Text einprogrammieren, wenn Sie das wollen.) • Erstellen Sie erst danach eine Funktion namens rechne(w, x, y, z), die x von

w abzieht, dann y dazu zählt und z wieder abzieht.

Kontrolle

Wo lag der Fehler im hier beschriebenen Programm? Und warum sollte Sie das verwundern, wenn Sie daran denken, dass es sich bei C um eine kompilierte Sprache handelt?

10.10 Wie eine Funktion vom Computer ausgeführt wird

Die meisten Handbücher sparen sich diesen Teil, dabei ist er immens wichtig, damit Sie bestimmte fortgeschrittene Programmiermethoden verstehen und anwenden können.

Sie wissen jetzt, wie Sie eine Funktion programmieren, doch was tut der Rechner eigentlich, wenn er eine Funktion ausführt?

Um es etwas anschaulicher zu bekommen, nennen wir die Stelle eines Programms, von der aus eine Funktion aufgerufen wird funktionsaufrufende Stelle. Das ist kein Fachbegriff, aber es wird im Folgenden hilfreich sein, wenn wir hierfür einen definierten Begriff haben.

Wie Sie wissen werden die Argumente einer Funktion als eigenständige Variablen im Speicher abgelegt, damit sie entsprechend der Funktion geändert werden können, ohne an den ursprünglichen Variablen etwas zu ändern. Der praktische Nutzen besteht darin, dass Sie die Möglichkeit haben, die ursprünglichen Variablen zu ändern, aber genauso die Freiheit haben, Sie beizubehalten.

Nehmen wir nun an, unsere Funktion ist so wie eine Subroutine bei der maschinennahen Programmierung programmiert. Dann wird die funktionsaufrufende Stelle im LR, dem Link Register gespeichert. Anschließend arbeitet die Funktion wie eine Subroutine mit dem übergebenen Argument ihre Programmzeilen ab, speichert den Übergabewert in einer Speicherstelle oder einem freien Register und anschließend wird das Programm nach der funktionsaufrufenden Stelle wieder fortgesetzt. Der Nachteil dieser Methode besteht darin, dass Sie im Grunde immer eine Subroutine vollständig abarbeiten lassen müssen, bevor Sie die nächste aufrufen können. Sie können nicht ohne weiteren Programmieraufwand Subroutinen aus Subroutinen aufrufen und jeweils individuell mit neuen Argumenten arbeiten lassen.

Funktionen können aber wesentlich komplexer eingesetzt werden: Sie können

aus einer Funktion heraus weitere Funktionen aufrufen und diese Aufrufe beliebig komplex staffeln. Und das auf eine Art und Weise, die im Gegensatz zu dieser Erklärung wie ein Kinderspiel ist. Im Gegensatz zur maschinennahen Programmierung, wo Ihnen lediglich ein Link Register zur Verfügung steht, wird hier bei jedem Funktionsaufruf quasi ein zusätzliches Link Register erzeugt. Außerdem wird jedes Argument, das bei einem solchen Funktionsaufruf übergeben wird eigenständig gespeichert.

In der Praxis bedeutet das, dass Sie alle Prozesse, die für sich abgeschlossen sind als eine Funktion programmieren können. Und auch wenn das bei kleinsten Programmen keinen großen Wert hat, ist es bereits bei Programmen mit vierzig oder mehr Programmzeilen eine unschätzbare Hilfe, um Übersichtlichkeit zu schaffen und mehrfachen Code zu vermeiden.

Eine ganz besonders mächtige Variante dieses Aufrufs von Funktionen aus anderen Funktionen lernen Sie in Kürze kennen. Die Rede ist von Rekursionen.

10.10.1 Dynamische Funktionen und Funktionen als Argumente von Funktionen

Auch wenn es in C keine dynamischen Funktionen gibt, sollten Sie sich mit diesem Thema auseinander setzen, weil es wichtig ist, dass Sie verstehen, was eine dynamische Funktion ist und wie Sie so etwas programmieren können. Das selbe gilt für Funktionen, die als Argument einer Funktion übergeben werden.

Sie wissen bereits, dass ein dynamischer Teil eines Programms während der Laufzeit (also zwischen Anfang und Ende eines Programms) geändert werden kann. Und bei nicht-statischen Variablen finden die meisten das auch einleuchtend. Aber wie sieht es bei dynamischen Funktionen aus? Da stellen sich den meisten die Nackenhaare auf und sie sagen, dass das doch unmöglich sein muss.

Sehen wir uns dazu nochmal an, wie eine Funktion programmiert wird: Sie definieren einen Namen, ggf. ein oder mehrere Argumente und den Datentyp eines Rückgabewertes. Und jetzt erinnern Sie sich daran, wie Sie eine Variable definieren: Sie legen einen Namen, sowie den Datentyp fest. Und jetzt fragen Sie sich selbst: Wenn es so leicht ist, den Inhalt einer Variablen zu ändern, warum sollten Sie dann nicht genauso leicht mittels einer Zuordnung den Rumpf einer Funktion austauschen? Bei C lautet die Antwort: Es ist eben so! (Was nicht gerade eine sehr befriedigende Antwort ist.)

Kommen wir nun zum zweiten Teil: Warum sollten Sie einer Funktion als

Argument eine andere Funktion übergeben wollen? Ganz einfach: Stellen Sie sich vor, Sie haben einen Taschenrechner programmiert, der einige grundlegende Funktionen berechnen kann. Wenn Sie nun im Laufe von Jahren immer mehr Funktionen einprogrammieren, dann wird Ihr Programm immer unübersichtlicher. Wenn Sie dagegen die ganzen zusätzlichen Funktionen in einem externen Programmteil ablegen könnten, ließe es sich deutlich übersichtlicher gestalten, ohne dass der Funktionsumfang reduziert würde.

10.11 Kontrollstrukturen

Bis jetzt können Sie in etwa dieselben Programme entwickeln wie bei der maschinennahen Programmierung, als Sie wussten, wie Sie Register und Speicher verwenden können. Doch während wir dort kaum über diese Möglichkeiten hinausgegangen sind, kommen wir jetzt zu einem der ersten Werkzeuge, die den Vorteil der imperativen gegenüber der maschinennahen Programmierung ausmachen: Mit einer imperativen Programmiersprache können Sie ganz leicht den Ablauf des Programms in Abhängigkeit von Bedingungen steuern.

Bei der maschinennahen Programmierung war das einzige leicht nutzbare Mittel hierzu das Branchen. Eine Möglichkeit zum Branchen bei imperativen Programmiersprachen haben Sie schon kennen gelernt: Dort nutzen wir Funktionen für genau diese Aufgabe. Ein zentraler Unterschied besteht allerdings darin, dass Sie beliebig Funktionen aus Funktionen heraus aufrufen können, ohne dazu besondere Maßnahmen treffen zu müssen.

Doch wenn wir über Kontrollstrukturen reden, dann meinen wir damit etwas wesentlich mächtigeres als die Ausführung einer Subroutine in Abhängigkeit vom Wert einer Variablen: Wir reden über eine beliebig komplexe Verschachtelung unterschiedlichster Bedingungen, denen jeweils gänzlich andere Programmausführungen folgen. Außerdem erreichen wir über sinnvoll benannte Funktionen, dass unser Code wesentlich besser lesbar ist. Das wiederum ermöglicht es uns, Fehler wesentlich schneller zu finden und zusätzlich, sie mit geringerem Aufwand zu korrigieren.

10.11.1 Wenn-Dann-Kontrolle

Die einfachste Kontrollstruktur (engl. control flow) können Sie einsetzen, wenn Sie den Programmablauf in Form von einfachen wenn-dann (engl. if-then) Bedingungen strukturieren können. Und ja, diese wenn-dann-Bedingungen können Sie mit fast beliebiger Tiefe staffeln. Aber wir beginnen zunächst mit dem einfachen Fall.

Stellen Sie sich vor, Sie wollten dazu die Algebra BROT in ein Programm umwandeln: Sie würden dann über Variablen konkrete Eigenschaften eines einzelnen Brotes definieren, weil Sie in C noch keine andere Möglichkeit kennen gelernt haben. (Um ein Brot als ein einzelnes Datenobjekt zu programmieren, ist eine objektorientierte Programmiersprache Voraussetzung.) Wenn Ihre Aufgabe nun darin besteht, eine Sortierung zu programmieren, die ähnliche Brote in irgendeiner Form versammelt, dann könnten Sie das in Form einfacher wenn-dann-Vergleiche tun.

Doch dafür benötigen wir zunächst einige Operationen aus dem Bereich der booleschen Logik, allgemein als Vergleichsoperatoren bekannt. Diese entsprechen zum Großteil den Symbolen, die Sie aus dem Mathematikunterricht kennen. (kleiner als: ¡, größer als: ¿, usw.) Einzig den Gleichheitsoperator müssen Sie sich gesondert merken, denn im Gegensatz zur Mathematik nutzen wir hier nicht ein einzelnes, sondern ein doppeltes Gleichheitszeichen. Wie Sie schon gelernt haben ist das einfache Gleichheitszeichen in Sprachen wie C der Zuordnungsoperator, durch den Sie einer Variablen einen Wert zuordnen.

• Prüfung auf Gleichheit von zwei Variablen: = = (zwei Gleichheitszeichen)

Diese Operatoren verwenden Sie genauso, wie Sie zuvor die einfachen Operatoren z.B. für die Addition verwendet haben, wo Sie anstelle von addiere a und b einfach a + b programmiert haben. Der Unterschied gegenüber den arithmetischen Operatoren besteht aber darin, dass das Ergebnis von Vergleichsoperatoren eine boolesche Variable ist. Und diese nutzen wir üblicherweise als anonyme Variablen. Kurz gesagt beschäftigt sich die boolesche Logik mit allen Fällen, in denen man zwischen wahr und falsch eindeutig unterscheiden kann.

Im Falle unserer wenn-dann-Kontrollstruktur können wir jetzt zum Beispiel unterschiedliche Programmabläufe in Abhängigkeit davon programmieren, ob ein Wert größer als ein anderer ist. Hier ein simples Beispiel in Pseudocode:

```
if ( a < b ) then { sortiere(a, b); }
else if (a > b) then { sortiere(b,a); }
else { printf(„Beide sind gleich.\); }
```

Aufgabe:

(1) Worin besteht der Unterschied zwischen dem eben aufgeführten und dem nun folgenden Algorithmus?

```
if ( a < b ) then { sortiere(a, b); }
else if (a == b) then { printf(„Beide sind gleich.\); }
else { sortiere(b,a); }
```

(2) Prüfen Sie mit einem C-Programm, ob (“Eins“ = = „Eins“) wahr ist. (Tipp: Sie können eine boolesche Variable nicht über printf() ausgeben lassen.

(3) Begründen Sie, warum ein solcher Vergleich bei manchen Sprachen wahr (true) und bei manchen Sprachen falsch (false) ist.

(4) Machen Sie sich bewusst, was geändert werden müsste, damit printf() eine boolesche Variable ausgeben kann und erweitern Sie Ihr Programm dann entsprechend. Wie schon bei einer früheren Aufgabe gilt hier wieder: Direkt ist das nicht möglich; Sie müssen sich eine Lösung für das Problem einfallen lassen, die Sie dann einprogrammieren müssen.

Aufgabe (schwer):

(1) Ist eine solche Wenn-Dann-Struktur eigentlich statisch oder dynamisch? (Die Begründung ist das wichtige.)

(2) Wenn Sie sich für eine der beiden Möglichkeiten entschieden haben: Was müsste gegeben sein, damit der andere Fall gilt?

Aufgabe (schwer):

Programmieren Sie die Funktion ggT(int a, int b), also die Funktion, die den größten gemeinsamen Teiler von a und b ausgibt.

Achtung: Diese Aufgabe ist deshalb schwer, weil Sie voraussetzt, dass Sie wirklich verstanden haben, welche Möglichkeiten Ihnen Funktionen bieten. Sollten Sie nach 15 Minuten Bedenk- und Probierzeit auf keine Idee gekommen sein, überspringen Sie die Aufgabe vorerst.

Kontrolle

Wann immer Sie bei einem Programm den Ablauf vom Zustand einzelner Variablen abhängig machen wollen, benutzen Sie einfache Wenn-Dann-Kontrollen. Bei C sehen die aufgrund Ihrer Entweder-Oder-Struktur etwas unübersichtlich aus, wenn hier viele Einzelfaktoren zur Entscheidung beitragen, aber Sie können natürlich mittels Funktionen mehr Eleganz in den Programmablauf bringen.

10.12 Rekursionen

Fragen Sie einhundert Studierende der Informatik, wo Sie zum ersten Mal Probleme beim Programmieren hatten und die Antwort wird lauten: Rekursionen.

Dabei sind Rekursionen eine unglaublich einfache Kontrollstruktur; Sie müssen nur (wie jeder ernstzunehmende Informatiker) wirklich verstanden haben, was eine Funktion ist. Das Wort Rekursion bedeutet übersetzt so viel wie „etwas, das immer wieder passiert“. Sie haben vielleicht schon von Schleifen gehört: Schleifen sind etwas, das Rekursionen ähnelt, aber im Gegensatz zu Rekursionen können Sie Schleifen nicht für jedes Problem nutzen, bei dem irgendein Programmteil sehr oft wiederholt werden soll. Für Schleifen gibt es den Fachbegriff der Iteration.

10.12.1 Rekursionen im Seminarraum

Bevor wir nun in die etwas formalere Erklärung eintauchen, was eine Rekursion ist und wie Sie sie programmieren können, zunächst ein anschauliches Beispiel für das, was bei einer Rekursion passiert:

Stellen Sie sich vor, Sie sitzen in einem Hörsaal und Ihnen ist langweilig. (Nein, Sie sollen es sich nur vorstellen... wehe, Ihnen ist wirklich langweilig.) In der Pause stellen sie fest, dass allen Ihren Kommilitonen langweilig ist und Sie vereinbaren ein Spiel: Jeder von Ihnen wird eine kleine Aufgabe erfüllen, wobei letztlich alle die gleiche Aufgabe haben. In der Programmierung würde man jetzt sagen, dass jeder von Ihnen eine Instanz einer Funktion ist: Sie sind alle unterschiedlich, sollen aber die gleiche Aufgabe erfüllen. Für Ihre Aufgabe benötigen Sie Stift und Papier. Beim Rechner wären dass die Register bzw. Adressen des Speichers.

Ihr Spiel (in der Programmierung wäre das der Algorithmus) nennt sich nun erzähle eine Geschichte und damit es lustig und überraschend wird, soll jeder von Ihnen nur ein Wort zur Geschichte hinzufügen. Hier hätten wir den Rumpf des Algorithmus bzw. den Rumpf des Programms in Pseudocode. Das könnte natürlich auch in einer Form notiert sein, die wie die Anleitung für ein Spiel verfasst ist, aber es ist wichtig, dass Sie lernen, Pseudocode und Programmcode wie normale Texte zu lesen.

```
erzaehleEineGeschichte(Wortform X)
{ wenn gilt, dass X = = Subjekt, dann
{ denke dir ein Subjekt aus;
notiere dieses Subjekt als Y;
```

```
X ist jetzt Prädikat;
}
wenn das nicht gilt, aber wenn gilt X = = Prädikat, dann
{  denke dir ein Prädikat aus;
notiere dieses Prädikat als Y;
X ist jetzt Objekt;
}
wenn beides nicht gilt (kurz: sonst), dann
{  denke dir ein Objekt aus;
notiere dieses Objekt als Y;
X ist jetzt Subjekt;
}
wenn jetzt gilt (du hast noch eine/n NachbarIn, der nicht mitgemacht hat)
{  fordere ihn/sie auf: erzaehleEineGeschichte(X);
merke dir seine/ihre Antwort als Z;
hänge Z an Y dran;
}
antworte dem-/derjenigen, die dir die Aufgabe erzaehleEineGeschichte() ge
}
```

Aufgaben: Schreiben Sie diesen Pseudocode so um, dass er soweit wie möglich wie ein C-Programm aussieht. Da Sie ja beispielsweise keinen Datentyp für Wortformen haben, tun Sie einfach so, als wenn das ein Datentyp wäre. Ähnliches gilt für Funktionen wie denke dir ein ... aus. Tun Sie hier einfach so, als wenn Sie eine Funktion hätten, die genau das tut.

Wie funktioniert dieses Spiel?

Und wie würde ein solcher Algorithmus im Speicher eines Computers aussehen?

Haben Sie schon eine Idee, was daran eine Rekursion ist?

10.12.2 Rekursionen etwas formaler

Eine Rekursion ist eine ganz normale Funktion, aber während die meisten Programmierer Funktionen im Grunde nur wie Subroutinen nutzen, kommt hier das volle Potenzial von Funktionen zum Einsatz: Sie werden nicht nur ein einziges Mal abgearbeitet, bevor sie einen Rückgabewert übergeben, sondern sie werden so lange einem aktualisierten Rückgabewert neu gestartet, bis eine bestimmte Bedingung erreicht ist. Erst dann wird der aktuelle Rückgabewert zurück gegeben.

Programmierer, die Rekursionen nicht vollständig verstehen werfen an dieser Stelle ein, dass das doch genauso ist wie bei einer Schleife, aber das ist falsch: Bei Schleifen ist diese Bedingung ein fester Wert, der zu Beginn des Schleifenaufrufs bekannt sein muss. Im Gegensatz dazu kann bei einer Rekursion die Bedingung bei jeder Wiederholung abgeprüft werden. Und diese Bedingung kann jede denkbare Bedingung sein; es muss kein Zahlenwert sein.

Um das zu verdeutlichen: Stellen Sie sich vor, Sie wollten ein Programm erstellen, das anhand eines Stadtplanes prüft, ob Sie innerhalb eine bestimmten Zeit zu Fuß ans Ziel kommen können. Keine Sorge, wir werden hier nicht im Detail erörtern, wie Sie so etwas programmieren können. Alles, was uns für den Moment interessiert ist die Frage, wie denn die Abbruchbedingung für eine Rekursion lauten würde, mit der wir ein solches Programm entwickeln würden. Und die wäre ja denkbar einfach: Computer, wenn du einen Weg gefunden hast, dann verrate ihn mir. Und jetzt dürfte auch dem letzten Schleifenvertreter klar sein, dass so etwas nicht mit einer Schleife programmierbar ist.

10.12.3 So programmieren Sie Rekursionen

Dann wollen wir einmal sehen, was Sie tun müssen, um Rekursionen zu programmieren...

Alles, was Sie für die Programmierung von Rekursionen beherrschen müssen sind wenn-dann-Kontrollen und Funktionen. Also haben Sie schon alle Kenntnisse an Bord, die Sie hierfür brauchen.

Das erste, was Sie benötigen, wenn Sie eine Rekursion entwerfen bzw. programmieren wollen ist die sogenannte Abbruchbedingung. Denn wie Sie gleich sehen werden ist eine Rekursion so strukturiert, dass Sie endlos laufen wird, wenn kein Kriterium einprogrammiert ist, durch das sie endet.

Nehmen wir einmal an, Sie wollen einen Zähler programmieren, der den Wert einer Variablen so lange erhöht, bis diese den Wert 100 erreicht hat. Diesen Zähler wollen wir nun in Form einer Rekursion realisieren, die wir als zaehler(int a) bezeichnen. Die Abbruchbedingung lautet hier a ¡ 100. Es wäre auch möglich, das Programm so zu entwickeln, dass die Abbruchbedingung a ¡ 101 lautet. Letztlich kommt es auf die Programmierung an.

Also sieht unsere Rekursion so aus:

```
int zaehler (int a)
{
```

```
if (a < 100) then
{
zaehler(a+1);
}
return a;
}
```

10.12.4 Darum funktionieren Rekursionen

Die meisten Einsteiger haben hier ein Problem, weil Sie denken, dass in der vierten Zeile (wo zaehler(a+1) aufgerufen wird) der Inhalt der Rekursion überschrieben würde. Das ist aber nicht so.

Deshalb werden wir uns ansehen, was passiert, wenn diese Rekursion mit dem Befehl zaehler(97); aufgerufen wird. Oben bei dem Beispiel mit den gelangweilten Studierenden haben Sie schon ein Beispiel dafür. Aber wir betrachten jetzt die Situation beim Programmablauf im Rechner. Dazu müssen Sie sich vor Augen halten, was der Computer (im Sinne der maschinennahen Programmierung) tut:

Beim Aufruf der Funktion wird eine Variable des Datentyps int erzeugt, was ja nichts anderes heißt, als dass (bei einem Cortex-M0) ein Speicherbereich von 32 Bit Länge für diese Variable reserviert wird. Nehmen wir an, dieser Speicherbereich wird an der Adresse 0x1000 reserviert. Der aktuelle Rekursionsaufruf bezeichnet diesen Speicherbereich mit a. Das ist deshalb kein Problem, weil er nichts davon weiß, dass alle anderen Rekursionsaufrufe ebenfalls einen Speicherbereich individuell mit a bezeichnen. Und die anderen Rekursionsaufrufe wissen ebenfalls nichts von den Bezeichnungen, die die anderen Rekursionsaufrufe benutzen. Erinnern Sie sich in diesem Zusammenhang bitte wieder an den Scope, denn über nichts anderes reden wir hier.

Nun wird in unserem Beispiel die Zahl 97, also hexadezimal 0x61 der neuen Variablen zugewiesen und somit unter der Speicheradresse 0x1000 gespeichert.

Nachdem nun die Funktion durch die Kontrollstruktur (if (97 ¡ 100)...) dazu aufgefordert wurde, die Funktion zaehler(97+1) auszuführen passiert folgendes: Die Funktion zaehler(int a) wird mit dem Argument 98 aufgerufen. Für uns heißt diese Funktion zwar genauso wie die letzte Funktion zaehler(int a), aber im Rechner werden die beiden Aufrufe getrennt voneinander im Speicher abgelegt. Es handelt sich also um zwei Instanzen einer Funktion. Stellen Sie sich das so vor, als wenn Sie einen Index verwenden würden:

Der Aufruf zaehler(97); wird im Rechner als der Aufruf zaehler1(97) gespeichert.

Die aus zaehler1(97) aufgerufene Funktion zaehler(98) wird intern als zaehler2(98) verwaltet und ist damit eben nicht derselbe Funktionsaufruf wie zaehler1(97).

Dieser Aufruf sorgt also nicht dafür, dass die Adresse 0x1000 überschrieben wird, sondern weil es ein neuer Funktionsaufruf ist, wird wieder eine neue Adresse für das Argument der Funktion reserviert. Hier wäre das also z.B. die Adresse 0x1004. Und an 0x1004 wird nun das aktuelle Argument, also der Wert 0x62 (hexadezimal für 98) gespeichert.

Dann folgt der nächste Aufruf von zaehler(int a) eben mit dem Wert 99. Das Ganze geht so lange weiter, bis die Kontrolle if (a ¡ 100) falsch ist. Und jetzt passiert folgendes:

Die aktuell aufgerufene Funktion zaehler(int a) (intern also zaehler4(100)) gibt nun den Wert 100 zurück. An wen dieser Wert zurückgegeben wird? Na an zaehler3(99), denn diese hatte ja zaehler(100) aufgerufen. Und was tut nun zaehler3(99)? Genau: Diese Funktion tut mit dem Rückgabewert von zaehler4(100) gar nichts und führt nur die verbliebene return-Zeile aus. Sie gibt also den Wert 98 an zaehler2(98) zurück.

Kontrolle

Überlegen Sie, was diese Rekursion ausgibt, wenn Sie „fertig“ ist. Das ist irgendwie nicht das, was wir uns vorgestellt haben, schließlich sollte sie doch bis 100 zählen. Da sollte Sie dann doch auch 100 ausgeben.

Lösen Sie deshalb zur Kontrolle das folgende Problem: Wie müssen Sie den Code anpassen, damit die Rekursion tatsächlich den Wert 100 ausgibt.

Keine Sorge, wenn Sie hier zunächst verzweifeln, das ist ganz normal. Dabei besteht die Lösung in zwei kleinen Anpassungen. Und wenn Sie die geschafft haben, dann beherrschen Sie eine der mächtigsten Kontrollstrukturen schlechthin.

Diese Aufgabe ist eine ganz einfache Rekursion, die Sie auch mit Hilfe einer Schleife hätten realisieren können. Deshalb folgt auch gleich eine Aufgabe, die nicht mit einer Schleife lösbar ist.

Aufgabe (schwer): Programmieren Sie eine Rekursion, deren Argument ei-

ne ganze Zahl (nennen wir sie einfach n) ist. Die Rekursion soll nun die Summe berechnen, die im Pascalschen Dreieck in der n'ten Zeile steht.

10.12.5 Schleifen

Bei vielen Programmierkursen werden Rekursionen übersprungen und anstatt dessen die sogenannten for- und while-Schleifen in allen Varianten besprochen. Das ist komplett überflüssig, weil Sie alles, was Sie mit einer for- oder while-Schleifen machen können wesentlich eleganter mit einer Rekursion erledigen können. Dazu kommt, dass Sie für jede Programmiersprache detailliert lernen müssen, wie Sie eine dieser Schleifen programmieren müssen.

Also lassen wir das doch lieber gleich ganz.

Na gut, wenn Sie es unbedingt wollen, können Sie von mir aus auch Schleifen benutzen. Suchen Sie einfach im Netz danach. Minuspunkte gibt's dafür nicht, aber Rekursionen müssen Sie in jedem Fall beherrschen.

10.13 Datenstrukturen

Denken Sie nochmal an die algebraische Struktur BROT. Und stellen Sie sich jetzt vor, Sie müssten verschiedene Elemente der algebraischen Struktur einzeln programmieren. Dann müssten Sie Unmengen an Variablen deklarieren und initialisieren. Das ist nicht nur sehr arbeitsaufwändig, es ist vor allem außerordentlich fehleranfällig. In objektorientierten Sprachen gibt es u.a. für diese Aufgabe die sogenannten Klassen, in C können wir uns lediglich der sogenannten Datenstrukturen bedienen, die aber auch schon ein recht mächtiges Mittel sind, um die Fehleranfälligkeit unseres Codes zu reduzieren.

Aber zunächst zur Frage, was eine Datenstruktur ist: Die naive Antwort lautet: Es ist eine strukturierte Methode, um Daten aufzubewahren. So naiv die Antwort, so wenig sagt Sie uns… Also versuchen wir es einmal anders: Stellen Sie sich vor, Sie hätten 95 Elemente, die eine konkrete Menge der Algebra BROT bilden. Oben hatten wir definiert, dass die folgenden Eigenschaften ein Element dieser Algebra auszeichnen: Knusprigkeit, Farbe, Temperatur, Form.

Wenn wir nur die Mittel zur Verfügung hätten, die Sie schon kennen gelernt haben, dann müssten wir jetzt die Variablen knusprigkeit1 bis knusprigkeit95, die Variablen farbe1 bis farbe95 usw. deklarieren und ihnen jeweils

einen zum Wert passenden Datentyp zuordnen.

Aber das brauchen wir. Wir können alternativ dazu eine Datenstruktur deklarieren. In Veranstaltungen zu Algorithmen und Datenstrukturen lernen Sie verschiedene dieser Datenstrukturen kennen. Wichtig ist, dass Sie in einer Sprache wie C jede Datenstruktur selbst einprogrammieren können, auch wenn sie nicht Teil der Sprache ist.

Für den Anfang genügt es, wenn Sie als Datenstruktur ein Array nutzen, mittel- bis langfristig sollten Sie allerdings lernen, wann Sie am besten welche Datenstruktur nutzen. Gerade verkettete Listen und Bäume sind in der Praxis häufig wesentlich effizienter und eleganter als ein Array, auch wenn sich Ihre Nutzung nicht ganz so einfach erschließt. Schlechte Programmierer und Neulinge erkennen Sie daran, dass diese in C ausschließlich Arrays als Datenstruktur nutzen.

10.13.1 Arrays

Eine Einschränkung, der ein Array in der Sprache C unterliegt, besteht darin, dass alle Elemente den gleichen Datentyp haben müssen. Dadurch lässt sich ein Array sehr leicht auf der Maschinenebene umsetzen. Dazu ein Beispiel:

Wenn Sie ein Array deklarieren, dass 95 Einträge hat und dessen Einträge alle vom Typ int sind, dann passiert im Hintergrund folgendes: Da jede Variable vom Typ int 32 Bit bzw. 4 Byte belegt, werden jetzt 95 * 4 Byte, also 380 Byte bzw. 380 Adressen am Stück für unser neues Array reserviert. (Sie wissen schon: Beim Programmieren in C sehen wir davon nichts.)

Im nächsten Schritt müssen wir noch jeden Eintrag des Array mit einem Wert initialisieren. Bei manchen Programmiersprachen passiert das automatisch, bei C nicht.

Schauen wir uns einmal an, wie eine Initialisierung praktisch durchgeführt wird: Angenommen wir initialisieren nur den fünfzigsten Eintrag eines Array mit dem Wert 27 (hexadezimal 0x1B). Dann wird zunächst geprüft, wie die Startadresse des Arrays lautet. Nehmen wir an, diese lautet 0x2000. Nehmen wir weiter an, es handelt sich um ein int-Array, dass also jeder Eintrag 4 Byte bzw. 4 Adressen belegt.

Dann wird nun die Startadresse genommen (0x2000) und dazu werden 49 * 4 (das ist 196, bzw. hexadezimal 0x124) addiert. Damit befindet sich der fünfzigste Eintrag des Array unter Adresse 0x2124. Jetzt wird also an der

Adresse 0x2124 der Wert 0x1B gespeichert.

Sie fragen sich, warum wir hier 49 * 4 und nicht 50 * 4 gerechnet haben? Ganz einfach: Da die Startadresse (hier abgekürzt als S, bzw. S + 0 * 4) die erste Adresse ist, an der Daten im Array gespeichert werden, ist S + 1 * 4 die Adresse, an der das zweite Element des Arrays gespeichert wird. Dementsprechend finden Sie das fünfzigste Element nicht an Adresse S + 4 * 50 sondern an Adresse

S + 4 * 49.

Die Initialisierung eines Array wird üblicherweise im Rahmen einer for-Schleife erledigt, aber wie Sie jetzt wissen, können Sie das in Form einer Rekursion erledigen.

Programmierung eines Array in C:

Für die Deklaration eines Arrays müssen Sie neben dem Datentyp der Einträge bei C von Beginn an wissen, wie viele Einträge das Array haben soll. Achtung: Sie dürfen hier von der Anzahl Elemente nichts abziehen, denn auch wenn das letzte Element die Nummer hat, die um eins kleiner als die Anzahl Elemente des Array ist, bleibt die Anzahl Elemente gleich.

In unserem Fall haben wir es also mit 95 Elementen im Array temperatur zu tun. (Sie wissen schon: Kann man Brot einfrieren?) Die Deklaration des Array sieht dann so aus:

```
int temperatur[95];
```

Denn im Gegensatz zu Funktionen aber genau wie eine Variable muss ein Array wieder deklariert werden, bevor es initialisiert werden kann.

Die Initialisierung eines Elements des Array ist wieder eine Zuweisung, die so ähnlich aussieht wie die Zuweisung eines Wertes zu einer Variablen. Nehmen wir an, wir wollen dem 35'igsten Element des Array einen Wert von 17 zuordnen, dann sähe das so aus:

```
temperatur[34] = 17;
```

Achtung: Vergessen Sie an dieser Stelle nicht, das wir bei der Zählung der Elemente eines Array bei 0 anfangen. Deshalb müssen wir hier nicht temperatur[35] den Wert 17 zuordnen, sondern wie soeben geschehen temperatur[34].

Aufgabe: Deklarieren und initialisieren Sie innerhalb eine C-Programms ein Array der Länge 20.

Die Elemente sollen vom Typ float sein. Das erste Element soll den Wert 10.0 haben.
Jedes Element soll mit einem Wert initialisiert werden, der um 1.23 größer ist als der seines Vorgängers. (Das erste Element bekommt also den Wert 10, das zweite den Wert 11.23, das dritte den Wert 12.46, usw.) Lassen Sie anschließend die Werte ausgeben, wobei nach jeweils zehn Werten ein Zeilenumbruch erfolgen soll.

Hinweis: Wenn Sie einen Fehler bekommen, bei dem so etwas wie array index out of bound steht, dann haben Sie einen Teil dessen ignoriert, was in diesem Abschnitt erklärt wurde.

Tipp: Erinnern Sie sich an den Modulo-Operator, dann können Sie sich einiges an Programmieraufwand sparen.

10.13.2 Strings

Diejenigen unter Ihnen, die schon ein wenig imperativ programmiert haben, werden bei der Einführung von Datentypen eingewandt haben, dass ein String doch ein Datentyp ist. Dort konnten Sie lediglich nachlesen, dass das falsch ist. Aber erst jetzt haben wir alles besprochen, was Sie wissen müssen, um zu verstehen, warum das falsch ist.

Denn ein String ist eine Datenstruktur. Genauer gesagt ist es ein Array vom Typ char und damit dynamisch.

Aufgabe:

Programmieren Sie ein solches char-Array, dass den Satz hello, world enthält und geben Sie den Inhalt des char-Array auf dem Bildschirm aus.

Statische Strings können Sie realisieren, indem Sie wie beim Quellcode am Anfang des Kapitels einen beliebigen Text zwischen doppelten Anführungszeichen setzen. („Die Antwort lautet: 42“)

Kontrolle

Ein auszugebener Text lautet Hallo, Weld (mit d statt t in Welt) und ist unter einem Bezeichner namens begruessung abgespeichert. Wie können Sie diesen Fehler korrigieren, wenn a) der Text als statischer String und b) als dynamischer String vorliegt?

10.13.3 Verkettete Listen, Bäume u.a.

Wenn Sie eine Veranstaltung wie Algorithmen und Datenstrukturen besuchen, fragen Sie sich nun, wie Sie denn beispielsweise eine verkettete Liste in C programmieren können? Das ist gar kein so großes Problem, allerdings kommen Sie hier um die Programmierung mit Pointern nicht herum. Über Pointer haben wir aber noch nicht gesprochen und werden das vorerst nicht tun, weil Sie zwar ein sehr mächtiges Mittel der Programmierung sind, aber im Grunde eine Einführung der maschinennahen Programmierung in die Sprache C. Zu diesem Zeitpunkt werden wir Sie deshalb noch nicht behandeln.

10.13.4 Zusammenfassung

Sie haben jetzt alles gelernt, was Sie für die grundlegende Programmierung in C benötigen. Einige Arten von Programmen können Sie mit diesem Wissen aber noch nicht erstellen. Damit Sie Klarheit haben, welche Arten von Programmen Sie jetzt noch nicht erstellen können, kommt hier eine kleine Aufstellung wichtiger Fälle: (So beißen Sie sich nicht die Zähne an einer Aufgabe aus, die Sie noch nicht lösen können.)

• Nutzereingaben: Sie wissen noch nicht, wie Sie Nutzereingaben z.B. über die Tastatur verarbeiten können. Wobei Ihnen hier nur ganz wenig Wissen fehlt. • Vernetzung: Sie wissen noch nicht, wie Sie ein C-Programm entwickeln sollen, das selbständig eine Verbindung über Netze aufbauen und nutzen kann. Selbst einfache Programme wie einen Instant Messenger können Sie also noch nicht programmieren. • Grafikbasierte Oberflächen: Sie wissen noch nicht, wie Sie grafische Elemente wie ein Menü oder Schaltflächen programmieren können.

Aber dennoch kennen Sie jetzt bereits alles, was Sie an Kernelementen der Sprache C benötigen, um ein beliebiges Programm für einen Cortex-M0 zu entwickeln. Jetzt kommen also die Bereiche, die nicht zum eigentlichen Kern der Sprache C gehören, die Sie aber innerhalb eines C-Programms benötigen, um einen Cortex-M0 zu programmieren.

Teil III

Fortgeschrittene und alternative Paradigmen

Kapitel 11

Klassenbasierte objektorientierte Programmierung

Kapitel 12

Funktionale Programmierung

Kapitel 13

Prototypbasierte objektorientierte Programmierung

Kapitel 14

Logische Programmierung

Teil IV

Maschinennahe Programmierung

Kapitel 15

Grundlagen der ARM Cortex-M0 Rechnerarchitektur

Kapitel 16

Grundlagen der Intel IAx86 Architektur