Ralf Salomon

Praktische Informatik und die Programmiersprache C

Eine Einführung
speziell für Elektrotechniker und andere Ingenieure

```
#include <stdio.h>

int main( int argc, char **argv )
    {
       printf( "Hi guys, welcome to the class!\n" );
    }
```

Ralf Salomon

Praktische Informatik und die Programmiersprache C
Eine Einführung speziell für Elektrotechniker und andere Ingenieure

ISBN 978-3-00-042684-1
2. korrigierte und erweiterte Auflage

Satz: Ralf Salomon
Druck und Verarbeitung: Westarp & Partner Digitaldruck Hohenwarsleben UG
Printed in Germany
Umwelthinweis: Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.

Email: `ralf.salomon@uni-rostock.de`
Web: `www.imd.uni-rostock.de/ma/rs`

Danksagung und allgemeine Referenzen

Die wesentlichen Teile des vorliegenden Manuskripts sind im Sommer 2012 entstanden und sind in erster Linie für unsere Studenten der Elektrotechnik gedacht und entsprechend auf ihren Kenntnisstand und ihre späteren Programmierbedürfnisse zugeschnitten.

Bei der Erstellung haben viele Personen geholfen. In erster Linie sind die Mitarbeiter Enrico Heinrich, Matthias Hinkfoth, Ralf Joost und Ralf Warmuth zu nennen, die durch ihre Korrekturen, Vorschläge, Hinweise und teils sehr intensive und kontroverse Diskussionen zum Gelingen maßgeblich beigetragen haben. Also Jungs, vielen Dank!

Desweiteren sind alle Studenten zu nennen, die dem Fehlerteufel zu Leibe gerückt sind. Großen Dank an Euch alle. Besonders hervorzuheben ist hier die Arbeit von Andrea Dorn, die das Skript noch einmal sehr gründlich gelesen und korrigiert hat.

Eine wesentliche Voraussetzung für das Schreiben dieses Manuskriptes war die sehr inspirierende Arbeitsumgebung, die ich im Sommer 2012 auf Maui vorfand. Daher gebührt großer Dank all den anonymen Beschäftigten folgender Lokalitäten, die mich unwissentlich vor allem durch Kaffee, Musik und ihren Spirit unterstützt haben: Denny's at South Kihei, Paia Bay Cafe at Paia, Starbucks at South Kihei und Starbucks at Kahului. Thanks guys, your help is highly appreciated!

Keine der hier vorzufindenden Textstellen und Bilder sind aus anderen als den im Text kenntlich gemachten Stellen übernommen worden. Natürlich kann es bei aller Sorgfalt sein, dass es in der Literatur ähnliche Textstellen oder Programme gibt. Doch sind diese Ähnlichkeiten reiner Zufall oder Resultat von mehr als 25 Jahre Programmier- und Lehrerfahrung, in denen man viel Erfahrung sammelt und vor allem auch Open-Source Quellen sieht, an die man sich heute nicht mehr erinnert.

Natürlich haben auch einige Bücher ihre unverkennbaren Spuren hinterlassen. Hier sind insbesondere das Buch über Algorithmen und Datenstrukturen, von Niklaus Wirth [5], die Einführung ins Software Engineering von Kimm et al. [3], der Klassiker über die Programmiersprache C von Kernighan und Ritchie [4] sowie das Buch über das Unix Betriebssystem von Maurice Bach [1] zu nennen.

Inhaltsverzeichnis

Kapitel 1

Vorwort

„Herzlichen Glückwunsch, Sie haben gut gewählt!“ So oder so ähnlich könnte ein Werbeblättchen anfangen. Aber hier handelt es sich nicht um eine Werbebroschüre, sondern um vorlesungsbegleitende Unterlagen für das sechsstündige Modul *Einführung in die Praktische Informatik* der Universität Rostock. Dementsprechend dienen diese Unterlagen auch nur zur *Unterstützung* beim Lernen und richten sich insbesondere an unsere Studenten der Elektrotechnik; bei diesen Unterlagen handelt es sich also nicht um ein eigenständiges Lehrbuch, das alle Aspekte der Programmiersprache C oder des systematischen Softwareengineerings behandelt.

„Das muss doch nun wirklich nicht sein, noch eine weitere Lehrunterlage zum Thema C!“ Ja, könnte man denken. Ein Blick in die Läden zeigt, dass bereits gefühlte 1000 Lehrbücher zu diesem Thema existieren. Aber so einfach ist die Sache nicht. Jedes Lehrbuch und jede Lehrveranstaltung richtet sich an eine spezielle Zielgruppe (Zuhörerschaft), hat einen festgelegten Zeitumfang, hat ihre eigenen Ziele und ihre eigenen Schwerpunkte. Vor diesem Hintergrund können Lehrunterlagen so gut sein wie sie wollen, sie nutzen wenig, wenn sich diese Aspekte nicht in den Unterlagen wiederfinden.

„Und wer will nun das hier lesen und lernen?“ Du! Denn sonst hättest du die Unterlagen nicht in Deiner Hand. Wie oben bereits angedeutet, richten wir uns mit unserer Lehrveranstaltung an Elektrotechniker, die sich im Grundstudium befinden. Entsprechend ist die gesamte Lehrveranstaltung auch genau auf die späteren Bedürfnisse dieser Zielgruppe zugeschnitten. Das heißt nicht, dass nicht auch andere Leser und Zuhörer von den Unterlagen profitieren können. Im Gegenteil, sie können, doch kann es gut sein, dass sich ihnen der Sinn einiger Aspekte nicht (unmittelbar) erschließt und ihnen damit diese eher als überflüssig erscheinen. Aber, mehr Wissen schadet ja bekanntlich nie.

„Cool, eine Lehrveranstaltung speziell für mich. Und was erwartet mich nun?“ Im Laufe ihres Studiums lernen Elektrotechniker viel über Hardware, Spannung und Strom, Logikgatter und deren Anbindung an die Umwelt. Insofern gehen wir davon aus, dass die Leser auch eine gewisse Affinität zu (low-level) Hardware haben und dass sie sich zumindest

später auf dieser Ebene wohl fühlen. Entsprechend verfolgen wir mit unserer Lehrveranstaltung auch eine Methodik, die darauf basiert, dass dem Programmierer die resultierende, grundlegende Abarbeitung jeder einzelnen Programmzeile auf Hardware-Ebene klar wird. Mit anderen Worten heißt das, dass später die Programme auf Basis der sicheren Beherrschung der Grundfertigkeiten und dem kreativen Zusammensetzen selbiger entstehen werden.

„Gibt's denn auch andere Ansätze?“ Ja! Viele Studenten der Informatik beispielsweise erhalten den Zugang zum Programmieren über virtuelle Rechner und abstrakte mathematische Modelle. Entsprechend fangen diese Studenten mit vielen Algorithmen an, die sie nur theoretisch behandeln und nicht auf einem Rechner implementieren und testen. Dies kommt oft erst später hinzu. Und über die Zeit entwickeln natürlich auch Informatikstudenten eine klare Vorstellung von der Arbeitsweise eines Computers.

„Und welcher Ansatz ist nun der bessere?“ Beide haben ihre Berechtigung. Informatiker arbeiten beispielsweise eher an der Entwicklung sehr großer Programme wie Datenbanksysteme, Informationssysteme, Compiler, Webserver etc. Elektrotechniker hingegen entwickeln oft ihre eigene Hardware, die sie zum Zwecke des Funktionsnachweises mal eben selbst programmieren müssen. Für dieses Ziel ist vor allem ein enger Hardware-Bezug von Nöten, weniger die Beherrschung ausgeklügelter Kompetenzen aus dem Bereich des Software Engineerings. Oder stark vereinfacht am Beispiel eines Hausbaus zusammengefasst:[1] Die Elektrotechniker sind wie Bauarbeiter, die Steine, Bretter, Sand und Kies zu einem Haus zusammenfügen, wohingegen Informatiker mit Architekten vergleichbar sind, die das Haus komplett planen, in Einzelteile aufteilen und am Ende hoffen, dass jemand kommt und ihnen das Haus auch baut.

„Und wie lerne ich jetzt am besten?“ Eine der besten Fragen! Die Lehrveranstaltung besteht aus sechs Komponenten: Erstens, in der Vorlesung werden grundlegende Konzepte erläutert, die ein Grundverständnis vermitteln sollen. Zweitens, zu den Vorlesungen werden einzelne Arbeitsblätter verteilt, die das Reflektieren des behandelten Stoffs unterstützen sollen. Drittens, der Vorlesungsstoff wird in den hier vorliegenen Unterlagen nebst der hier erwähnten Literatur vertieft und erweitert. Um das Maximum aus den Vorlesungen herauszuziehen, sollten diese Unterlagen auf jeden Fall nach, aber vorzugsweise auch vor den Vorlesungen gelesen werden. Viertens, für alle Stoffteile haben wir viele Übungsaufgaben formuliert, die dem *praktischen* Einüben des Stoffes dienen. Die Bearbeitung dieser Aufgaben ist von höchster Wichtigkeit. Die Erfahrung lehrt, dass diejenigen Studenten, die diese Übungsaufgaben weitestgehend selbstständig bearbeitet und verstanden haben, in der Regel auch gute bis sehr gute Prüfungsleistungen erreichen; diejenigen Studenten, die die Übungen ignorieren, abschreiben oder von anderen bearbeiten lassen, fallen in der Regel durch. Fünftens, organisieren wir einen recht umfangreichen Übungsbetrieb, in dem die Doktoranden mit Rat und Tat zur Seite stehen. Die Unterstützung geschieht auf individueller Basis und beinhaltet keinen Frontalunterricht, in dem weiteres Wissen präsentiert

[1] Diese Zusammenfassung ist selbstverständlich nur ein Bild und nicht mit tierischem Ernst zu interpretieren.

wird. Sechstens, stellen wir auch „Musterlösungen“ bereit, die nach Möglichkeit *nach* der eigenen Programmerstellung konsultiert werden sollten, um kommentierte Alternativen zu sehen und dabei nach Möglichkeit die eigenen Kompetenzen zu erweitern.

„Wo anfangen?“ Aller Anfang ist schwer, besonders beim Programmieren. Man kann leider nicht in irgendeiner Ecke anfangen und sich dann dem Thema langsam nähern. Die Zusammenhänge zwischen Hard- und Software sind sehr komplex, sodass man immer mitten drin steckt, egal, wo und wie man beginnt. Das einzige ist, dass man versuchen kann, eine Ecke zu finden, in der das Chaos nicht ganz so groß ist. Aber so oder so ist der Komplexitätsgrad sehr hoch und man muss wohl oder übel einiges glauben und akzeptieren, dass man anfänglich nicht alles versteht; dieses Verständnis kommt mit der Zeit. Und irgendwann versteht man fast alles.

„Und was erwartet mich nun?“ Wir haben diese Unterlagen mit großer Sorgfalt erstellt und versucht, möglichst viele Inhalte aufzunehmen. Aber natürlich mussten auch wir uns auf diejenigen Inhalte konzentrieren, die für Elektrotechniker wichtig und/oder sinnvoll sind *und* im Rahmen einer sechsstündigen Veranstaltung bewältigt werden können. Die einzelnen Kapitel sind nun so aufgebaut, dass sie nach Möglichkeit nur einen isolierten Aspekt behandeln. Dadurch werden sie einschließlich der präsentierten Beispiele kurz und überschaubar. Dies hat natürlich eine höhere Zahl von Kapiteln zur Folge. Die einzelnen Kapitel sind nun so angeordnet, dass die notwendigen Voraussetzungen immer zuerst in gesonderten Kapiteln behandelt werden, damit innerhalb eines Kapitels keine Exkurse notwendig sind. Dies schlägt sich direkt in der Organisation der einzelnen Kapitel nieder, sie bestehen im Regelfall nur aus einfachen Unterkapiteln, die nicht in weitere Unterkapitel unterteilt sind. In der heutigen Terminologie würde man vielleicht sagen, dass die Organisationsstrukur sehr flach gehalten ist.

So, dann mal frisch an’s Werk und viel Spass beim Programmieren.

Kapitel 2

Motivation: Warum das alles?

Erste Antwortversuche: Schon in der ersten Vorlesung beschleicht die meisten Studenten ein ungutes Gefühl und sie fragen sich: „Warum das alles?“ Diese Frage ist sehr berechtigt! Eine ersten Antwort könnte lauten: „Ja, die Dozenten können es, also müssen wir Studenten da auch durch.“ So plausibel diese Antwort auch sein mag, so falsch ist sie.

Eine zweite Antwort könnte lauten: „Weil das Programmieren in C im Lehrplan steht, quälen sie uns damit.“ Als Antwort gar nicht mal so schlecht, doch trifft sie den Kern noch nicht so richtig. Doch wenn man sich jetzt fragt, warum Programmieren überhaupt auf dem Lehrplan eines Elektrotechnikers steht, kommt man den richtigen Antworten näher.

Am Anfang ihres Studiums denken viele Studenten der Elektrotechnik, dass das Programmieren für sie von sehr untergeordneter Bedeutung ist; gerade deshalb haben sie ja Elektrotechnik und nicht Informatik gewählt. Doch genau hier ist eines der wesentlichen Probleme versteckt: auch für den Elektrotechniker besteht mindestens die Hälfte seines beruflichen Daseins aus Programmieren.

Natürlich sind bezüglich des Programmierens die Anforderungen, Aufgaben und Tätigkeiten ganz andere als bei den Informatikern. Aber sie sind da und bestehen aus Dingen wie dem Programmieren von HTML-Seiten, Word, Excel und dem Programmieren selbstentwickelter Hardware, Platinen oder (eingebetteten) Systemen. Und genau hier kommen die Vorzüge der Programmiersprache zum tragen: C unterstützt das Hardware-nahe Programmieren und damit die schnelle und effektive Realisierung kleiner Programme zum Testen, Evaluieren oder Betreiben der eigenen Hardware.

Elektrotechniker werden von befreundeten Informatikern des Öfteren gefragt, warum sie diese altmodische Programmiersprache C lernen und nicht lieber gleich zu Java oder C++ wechseln, denn diese Sprachen seien doch wesentlich leistungsfähiger. Natürlich stimmen die Aussagen bezüglich der Leistungsfähigkeit, doch für die schnelle Implementierung eines Hardware-nahen Programms ist C immer noch en vogue, da C auf die Anforderungen von Hardware-Achitekturen zugeschnitten ist.

Die bisherigen Ausführungen kann man wie folgt zusammenfassen: Ein Elektrotechniker kommt an der (hemdsärmligen Nutzung) der Programmiersprache C nicht vorbei und je besser er sie kann, um so leichter kann er damit leben.

So weit C auch verbreitet ist, so ist das Erlernen dieser Programmiersprache mit einer Reihe von Problemen verbunden, die einem Studenten den letzten Nerv rauben können; dies liegt aber am Sprachdesign und nicht an den Lernenden. Seinerzeit wurde C von Programmierexperten für andere Programmierexperten entwickelt. Das bedeutet: die Designer von C gingen davon aus, dass C-Programmierer wissen was sie tun und bereits das notwendige Rüstzeug aus dem Bereich der Programmierungstechniken besitzen. Aber genau das ist logischerweise bei Programmieranfängern noch gar nicht vorhanden.

C ist so ausgelegt, dass der Fachmann sehr kompakt, das heißt mit sehr wenig Tipparbeit seine Programme erstellen kann. Ferner ist die Programmiersprache C so gestaltet worden, dass man mit ihr so gut wie alles machen kann. Auch das ist erst mal gut, doch ist es gleichzeitig eine wesentliche Fehlerquelle: Aus Sicht der Programmiersprache sind die erstellten Programme völlig korrekt, doch verhalten sie sich anders, als der Programmieranfänger gedacht bzw. gehofft hat. Und genau das kostet Nerven, da es die Fehlersuche so mühsam macht. Hier gibt es wenige gute Ratschläge; Programmieranfänger sind aber gut beraten, sich folgende *Verhaltensweisen* anzugewöhnen:

1. Sehr sorgfältig beim Programmieren vorgehen und eine mentale Einstellung wie beispielsweise „ich weiss zwar nicht wie, aber irgendwie wird es schon funktionieren“ unbedingt vermeiden.
2. Man sollte sich sehr im Klaren darüber sein, wie die einzelnen Datenstrukturen und Anweisungen vom Rechner verwaltet und abgearbeitet werden.
3. Alle Programmteile sollten (zumindest anfänglich) schrittweise entwickelt und jeweils sehr gründlich getestet werden.

Doch leider halten sich die wenigsten Programmieranfänger an diese „schlauen“ Ratschläge.

Zusammenfassung: Alles Gesagte kann man vielleicht wie folgt zusammenfassen: Das Erlernen der Programmiersprache C ist nicht nur mühsam sondern auch geprägt von Fallstricken, Merkwürdigkeiten und nahezu unendlich vielen Programmabstürzen. Aber in der Hand eines routinierten Fachmanns ist diese Programmiersprache ein sehr mächtiges Werkzeug, mit dem man sehr viel und sehr schnell realisieren kann. Insofern hat es sich für die Fachleute gelohnt, durch dieses Tal der Tränen zu marschieren.

Teil I

Der Trailer

Kapitel 3

Ein paar Vorbemerkungen zum Gesamtabriss

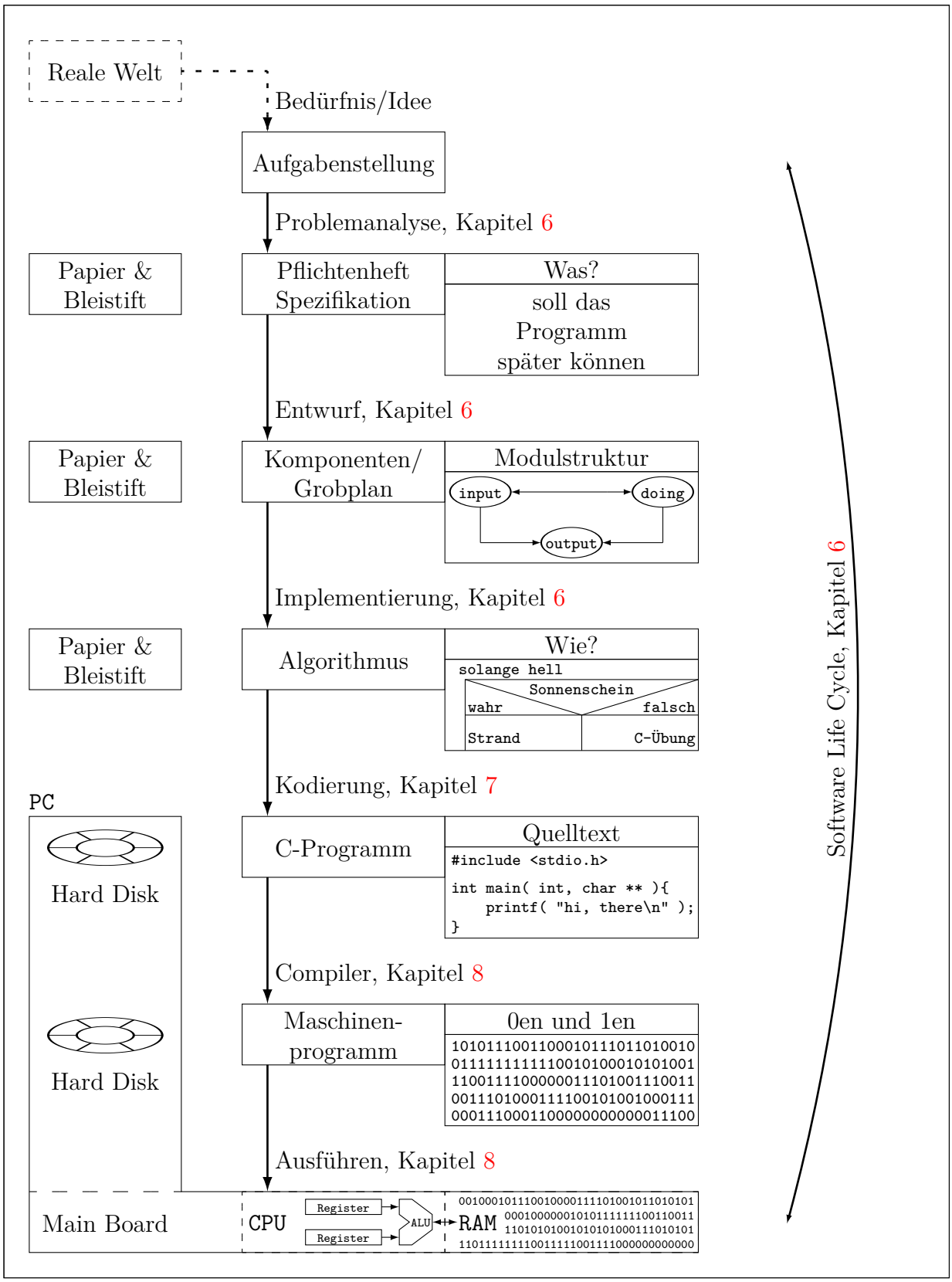
„Was ist denn nun eigentlich Programmieren?“ Viele von Euch werden sich darunter wohl einen Cola trinkenden und Chips essenden, leicht übergewichtigen Jungen vorstellen, der Nächte lang vor seinem PC sitzt und stundenlang irgendwelchen Programm-Code eintippt. Ganz falsch! Programmieren ist viel mehr. Man versteht darunter den gesamten Prozess von der ursprünglichen Idee bis hin zum fertigen Programm mit all seinen Tests sowie späteren Wartungs- und Änderungsarbeiten.

Je nach Phase und Bearbeitungsebene werden verschiedene Dokumente erstellt und kommen unterschiedliche Methoden bzw. Werkzeuge zum Einsatz. Bei dieser Fülle ist es für den Studenten wahnsinnig schwer, den Überblick zu behalten und nachzuvollziehen, wo denn die Dozenten gerade sind und was sie von einem wollen. Um hier eine *Navigationshilfe* zu erhalten, haben wir einmal versucht, alles in ein kleines, stark vereinfachtes Bildchen zu packen, das Ihr auf der nächsten Seite findet. Dieses Bildchen wird uns in der einen oder anderen Form in den weiteren Skriptteilen immer wieder begegnen, damit jeder weiß, wo wir gerade sind und was wir besprechen wollen. Je nach Zielstellung der einzelnen Kapitel werden immer weitere Details hinzukommen.

In diesem ersten Skriptteil werden wir einmal kurz und notgedrungen oberflächlich durch alle Ebenen hindurch gehen und dabei die wichtigsten Konzepte[1] kurz erwähnen. Trotz aller Oberflächlichkeit will Euch dieser Skriptteil vermitteln, *wo wir her kommen* und *wo die Reise hin gehen soll.* Die in der Abbildung eingetragenen Kapitel verweisen auf die entsprechenden Stellen dieses ersten Teils.

Wie im Bild zu sehen ist, fängt meist alles mit einer ersten Idee und/oder wachsendem Bedürfnis an. Auf dieser Ebene werden die Ideen in der Regel sehr vage beschrieben.

[1] Die meisten dieser Konzepte bieten übrigens genügend Stoff für eine eigenständige drei- oder sechsstündige Vorlesung.


Reale Welt
Bedürfnis/Idee
Aufgabenstellung
Problemanalyse, Kapitel 6
Papier & Bleistift
Pflichtenheft Spezifikation
Was?
soll das Programm später können
Entwurf, Kapitel 6
Papier & Bleistift
Komponenten/ Grobplan
Modulstruktur
input
doing
output
Implementierung, Kapitel 6
Papier & Bleistift
Algorithmus
Wie?
solange hell
Sonnenschein
wahr
falsch
Strand
C-Übung
Kodierung, Kapitel 7
PC
Hard Disk
C-Programm
Quelltext
#include <stdio.h>
int main(int, char **){
printf("hi, there\n");
}
Compiler, Kapitel 8
Hard Disk
Maschinen-programm
0en und 1en
Ausführen, Kapitel 8
Main Board
CPU
Register
Register
ALU
RAM
Software Life Cycle, Kapitel 6

Beispielsweise könnte jemand auf die Idee kommen, ein neues Haus für eine vierköpfige Famile zu bauen oder die Informatikinfrastruktur für einen neuen Hauptstadtflughafen zu konzipieren (*Just kiddin'*).

Das Ziel des nun folgenden Prozesses ist die Erstellung einer Software (eines Programms), das vom Prozessor (CPU) ausgeführt werden kann. Im Bild ist diese Ebene ganz unten dargestellt. Um den Aufbau und die Herangehensweisen der dazwischen liegenden Prozesse halbwegs zu verstehen, müssen wir zuerst wissen, was ein derartiger Prozessor kann und was er eben nicht kann. Dazu wird zuerst in Kapitel 4 ein Rechner (PC) zerlegt. Anschließend schaut Kapitel 5 etwas genauer auf die beiden Kernelemente der Programmausführung, nämlich die CPU und den Arbeitsspeicher (als RAM gekennzeichnet).

Zwischen diesen beiden Endpunkten, also der ersten vagen Idee bis hin zur Verwendung eines Programms auf einem Rechner, befinden sich gemäß des Bildes sechs Ebenen[2], die im Rahmen der Programmierung auch als Software Life Cycle[3] bezeichnet werden. Die Kernidee des Software Life Cycles ist es, eine halbwegs strukturierte Herangehensweise vorzuschlagen, mit deren Hilfe die Erfolgsquote für ein sinnvolles und korrektes Programm möglichst hoch ist. Die einzelnen Arbeitsschritte sind also kein Muss oder keine Gängelei sondern als Hilfestellung für das möglichst effiziente Arbeiten gedacht. Die ersten vier Arbeitsebenen dieses Software Life Cycles werden in Kapitel 6 kurz erläutert. Wichtig zu verstehen ist, dass die dort vorgestellten Methoden unabhängig von einer gewählten Programmiersprache sind und eher „normalen" Ingenieursmethoden entsprechen.

Die Erläuterungen von Kapitel 6 werden von der Entwicklung eines kleinen Programms zur Berechnung der Fläche eines Rechtecks begleitet. Der dabei im vierten Arbeitsschritt entstehende Algorithmus ist (weitestgehend) programmiersprachenunabhängig und wird erst jetzt, im fünften Arbeitsschritt, in eine konkrete Programmiersprache umgesetzt. Dies ist ziemlich direkt machbar und wird in Kapitel 7 anhand eines ersten C-Programms gezeigt.

Bevor dieses Programm von der CPU ausgeführt werden kann, muss es noch in den Maschinencode der unten liegenden CPU umgewandelt werden. Dieses Umwandeln wird vom Compiler erledigt. Anschließend kann es gestartet werden :-) ! Die beiden Arbeitsschritte, Compilieren und Programmstart, werden in Kapitel 8 erklärt.

Abgeschlossen wird dieser erste Skriptteil durch ein kleines *Preview*, um Euch das Bearbeiten der nächsten Übungsaufgaben auch ohne Studium der weiteren Skriptteile zu ermöglichen.

[2] Je nach Zielstellung und Hintergrundwissen werden in der Literatur auch andere Beschreibungen präsentiert; diese Darstellung ist nur eine von mehreren Möglichkeiten.

[3] Das englische Wort *cycle* bedeutet nicht nur Kreis oder Zyklus sondern auch Ablauf und Arbeitsablauf.

Kapitel 4

Aufbau eines PCs

Ein Blick in die Regale der Elektronikmärkte und Online-Shops erweckt den Eindruck, dass Computer (Personal Computer (PCs)) in ganz unterschiedlicher Form daher kommen. Es gibt Workstations, Desktops, Notebooks, Netbooks, Ultrabooks, PDAs, Tablets und Smartphones, um einfach mal ein paar wenige zu nennen. Äußerlich betrachtet sehen sie alle verschieden aus. Es ist immer eine flache „Schachtel“, manchmal mit einer zweiten dran, die wie ein Fernseher aussieht, manchmal noch mit einer dritten, die einer Schreibmaschinentastatur ähnelt. So wichtig wie diese Äußerlichkeiten für Aspekte wie Marketing, Preis, Einsatzgebiet usw. sein mögen, so unwichtig sind sie für die Funktionsweise, denn eigentlich funktionieren sie alle nach den selben Prinzipien. In diesem Kapitel gehen wir ein wenig auf den Aufbau dieser PCs ein, beschreiben kurz die wichtigsten Hard- und Software-Komponenten und erläutern, was eigentlich das Betriebssystem ist und was es macht.

4.1 Der erste Einblick

Wir stellen uns einfach mal vor, dass wir einen handelsüblichen PC nehmen und diesen aufschrauben. Was sehen wir da? Wir sehen ein offenes Gehäuse, in dem sich viele bunte Kabel befinden, die einzelne Kleinteile, Baugruppen sowie Stecker und Buchsen miteinander verbinden.

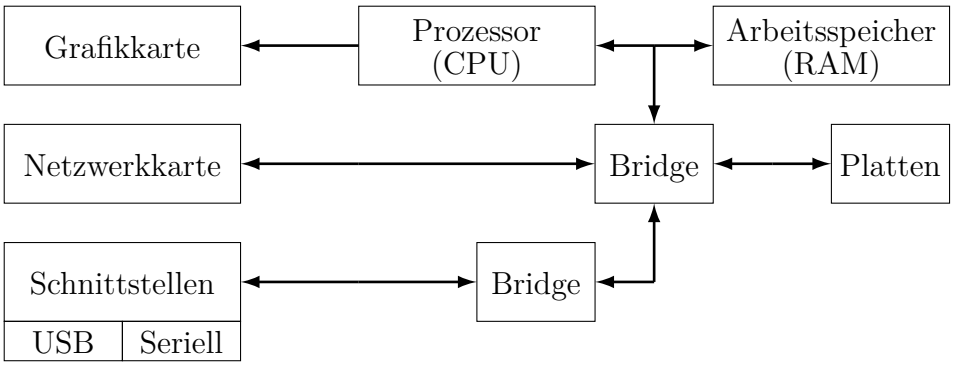
„Warum sind die ganzen Kabel so bunt? Welche Bedeutung hat das?“ Die Kabelfarbe ist ohne jegliche Bedeutung. Durch sie weiß der Techniker, was wo hin gehört. *„Und warum sind manche dünner und andere dicker?“* Je nach Dicke eines Kabels kann dort mehr oder weniger Strom durchfließen. Entsprechend sind die dicken Kabel in der Regel für die Stromversorgung, die dünnen für die Kommunikation der Baugruppen untereinander.

„Sind denn nun Notebooks etc. anders aufgebaut?“ Nein, all diese Teile sind im Grunde genommen gleich aufgebaut. Sie unterscheiden sich häufig nur in der Größe der Baugruppen,

deren Anzahl und deren Leistungsfähigkeit. Aber von der funktionalen Seite her betrachtet, sind sie alle mehr oder weniger gleich.

4.2 Die Top-Level Hardware-Architektur

Wenn wir uns jetzt von den einzelnen Baugruppen und Platinen (das sind die Teile, auf denen sich die Bauteile befinden) lösen, bekommen wir ein Top-Level Hardware-Architektur Modell. Dieses Modell beschreibt, welche Funktionen in einem Rechner zusammenspielen. Ein einfaches Modell sieht wie folgt aus:



In diesem Bild symbolisieren die einzelnen Kästchen einzelne Funktionsgruppen, die miteinander kommunizieren, was mittels der schwarzen Linien nebst Pfeilen illustriert ist. Als erstes besprechen wir die Funktion der einzelnen Blöcke im „täglichen“ Arbeitsablauf:

Schnittstellen: Das ist mal das Einfachste. Hier werden unsere Peripheriegeräte wie Maus, Beamer, Memory Stick etc. angeschlossen. Von hier geht es über spezielle Hardware-Bausteine (die sich hinter dem Wort „Schnittstellen“ befinden) zum Prozessor.

Netzwerkkarte: Auch dieses Teil sollte den meisten bekannt sein. Hierüber wird in der Regel die Verbindung zum Internet hergestellt. Dies kann über Kabel oder auch drahtlos geschehen.

Platten: Hier werden alle Daten permanent gespeichert. Bei vielen Geräten handelt es sich tatsächlich immer noch um magnetisierbare, rotierende Scheiben, auf die Daten aufgebracht und wieder herunter gelesen werden können. In modernen Geräten befinden sich SSDs (Solid State Disks), die den Plattenspeicher mittels Halbleiterelementen realisieren, wie sie auch in den üblichen MP3-Spielern und Memory-Sticks zu finden sind. Auf diesen Datenplatten befinden sich alle Dateien, die man so benötigt. Dazu zählen Bilder, Musikstücke, pdf-Dateien, word-Dokumente und alles mögliche, was man sich so aus dem Internet herunter lädt. Neben diesen „Daten Dateien“ befinden sich auch alle Programme auf diesen Platten. Mit anderen Worten: Wenn wir ein wenig herumsuchen, finden wir irgendwo alle Programme wie `word`, `excel`

und `firefox`. Und natürlich alle anderen Programme auch. Zusammengefasst kann man sagen: Die Plattenlaufwerke beherbergen alle Daten und Programme, die man längerfristig benötigt.

Bridge: Dabei handelt es sich um ein spezielles Hardware-Bauteil, das die Daten und Kommunikationsprotokolle aneinander anpasst. Dies beinhaltet auch die Übertragungsgeschwindigkeiten der einzelnen Baugruppen. Beispielsweise ist ein Plattenlaufwerk wesentlich langsamer als der Prozessor. Daher werden einzelne Daten in der Bridge zwischengespeichert und blockweise an den Prozessor bzw. den Arbeitsspeicher (RAM) übertragen, um den Prozessor nicht unnötig oft zu stören und von seiner Arbeit abzuhalten.

Grafikkarte: Die Grafikkarte stellt die Verbindung zwischen Prozessor und Bildschirm her. Alle Bildschirmausgaben wie Texte, Bilder, Grafiken etc. werden vom Prozessor in einem geeigneten Format in die Grafikkarte geschrieben. Die Grafikkarte ihrerseits sorgt dann selbstständig dafür, dass die eingeschriebenen Daten auf dem Bildschirm ausgegeben werden.

Prozessor (CPU) und Arbeitsspeicher (RAM): Diese beiden Komponenten sollte man zusammen betrachten. Aber erst mal zu den Abkürzungen: CPU bedeutet Central Processing Unit und ist das englische Wort für (Zentral-) Prozessor. RAM bedeutet Random Access Memory und bezeichnet in der Regel den Arbeitsspeicher, auch wenn diese Abkürzung nicht ganz korrekt ist.

Die CPU ist nun diejenige Instanz, die diejenigen Dinge abarbeitet, die langläufig (nicht ganz korrekt) als Programme bezeichnet werden. Hier geht also alles durch, jeder Maus-Klick, jeder Tastendruck, einfach alles. Um dies zu bewerkstelligen *muss* das Programm zuvor vom Plattenlaufwerk in den Arbeitsspeicher geladen werden. Von dort aus wird das Programm Anweisung für Anweisung abgearbeitet. Sollten mehrere Programme gleichzeitig laufen, beispielsweise `word`, der Browser und ein bisschen Musik im Hintergrund, sind diese Programme (oder zumindest Teile davon) auch gleichzeitig im Arbeitsspeicher. Das Betriebssystem sorgt dafür, dass sich die Programme nicht in die Quere kommen.

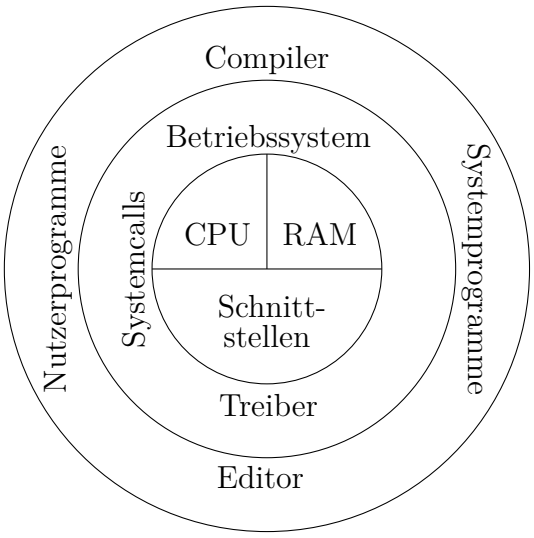
Das eben beschriebene Modell ist nur *eine* Möglichkeit von vielen. Die einzelnen Rechner unterscheiden sich in der Zahl der Schnittstellen, den Verarbeitungsgeschwindigkeiten, der Anzahl und Größe der Plattenlaufwerke, der Zahl der Netzwerkkarten etc. Auch die interne Struktur variiert von Rechner zu Rechner. Ebenso hängt es vom Rechner ab, welche Funktionsblöcke sich in welchen Bauteilen und auf welchen Boards befinden. Aber trotz all dieser Unterschiede: die generelle Funktionsweise ist bei allen gleich.

Im Rahmen unserer Lehrveranstaltung sind nur CPU und RAM von Interesse; alle anderen Komponenten sind zwar für die Funktionsweise des Gesamtsystems notwendig, stehen aber nicht in unserem Fokus und werden von uns auch nicht weiter behandelt.

4.3 Das Betriebssystem

Das Wort „Betriebssystem“ hat schon fast jeder einmal gehört. Die meisten wissen wohl, dass Windows ein Betriebssystem ist. Viele können sich dafür nicht vorstellen, dass es auch andere Betriebssysteme als Windows gibt. Hierzu gehört beispielsweise das Betriebssystem Linux, das gratis verwendet werden kann. Die Frage ist nun, was ist eigentlich das Betriebssystem und was macht es? Die klassische Sichtweise auf diese Frage ist in folgendem Zwiebelmodell wiedergegeben:

Im klassischen Zwiebelmodell unterscheidet man drei Schichten. Die innerste Schicht besteht aus der Hardware, einschließlich CPU, RAM und aller Schnittstellen. Darum herum befindet sich das Betriebssystem, das unter anderem die Treiber für die einzelnen Hardwarekomponenten beinhaltet. Der Zugriff auf die Funktionalitäten des Betriebssystems erfolgt über definierte Schnittstellen, die man System Calls nennt. In der äußersten Schicht befinden sind die System- und Dienstprogramme sowie die vom Benutzer erstellten eigenen Programme. Der Compiler ist eines der wichtigen Dienstprogramme. Es übersetzt Quellcode einer höheren Programmiersprache in den Maschinencode der CPU, worauf wir im nächsten Abschnitt etwas näher eingehen.



Nach diesem Modell ist also das Betriebssystem für alle betrieblichen Abläufe innerhalb des Systems verantwortlich. Durch die *Kapselung* der Hardware durch das Betriebssystem stellt dieses einen Schutzmechanismus dar, der von fehlerhaften Nutzerprogrammen nur sehr schwer durchbrochen werden kann. Zumindest in älteren Windows Versionen war diese Kaspelung nicht so stark, sodass leicht fehlerhafte Nutzerprogramme zu einem kompletten Systemabsturz führen konnten. In Unix-artigen Betriebssystemen ist diese Kapselung traditionell sehr stark, sodass solche Systeme in der Regel recht lange laufen, ohne das ein Re-Boot notwendig ist.

Eine weitere Frage ist, inwiefern eine grafische Nutzeroberfläche[1] zum Betriebssystem gehört oder eben nicht. In Linux-artigen Betriebssystemen ist die grafische Oberfläche eine eigenständige Komponente, die in derartigen Systemen in verschiedenen Variationen verfügbar ist. Beispiele sind `gnome`, `KDE` und `IceWM`. In Windows ist die grafische Oberfläche fest in das Betriebssystem integriert und somit integraler Bestandteil, der nicht durch andere Varianten ausgetauscht werden kann, was unter anderem auch zu gerichtlichen Auseinandersetzungen geführt hat.

[1] Das Wort „Nutzer-“ bzw. „Benutzeroberfläche“ ist semantisch nicht ganz korrekt und heißt daher im Fachdeutsch *Benutzungsoberfläche*.

4.4 Was ist ein Programm?

„Was ist ein Programm, wo befindet es sich, wie wird es gestartet, wann ist ein Programm ein Programm?“ Es ist recht schwer, diesen Fragenkomplex einem Programmieranfänger halbwegs zufriedenstellend zu beantworten. Dennoch versuchen wir es hier einmal wie folgt:

Was sind Programme, Programmiersprachen und Compiler? Häufig werden diese Begriffe nicht klar genug voneinander getrennt. Programme werden fast immer in höheren Programmiersprachen wie C, Pascal und Modula-2 entwickelt. Häufig wird dieser Quell*text* fälschlicherweise Programm genannt. Aber es sind nur Daten, die vom Compiler (Übersetzer) in Maschinencode übersetzt werden. Auch dieser Maschinencode ist ein Haufen von Daten, die erst noch ein Programm werden wollen.

Wann ist ein Programm ein Programm? In obigem Hardware-Modell gibt es zwei Orte, an denen sich größere Mengen von Bits und Bytes befinden können. Diese sind der Arbeitsspeicher (RAM) und die Plattenlaufwerke, wobei letztere um ein Vielfaches größer sind als ein RAM. Prinzipiell kann sich ein Programm an beiden Orten befinden. Im engeren Sinn ist ein Programm nur dann ein Programm, wenn es die drei folgenden Bedingungen erfüllt:

1. Es muss in Maschinensprache vorliegen
2. Es muss sich im Arbeitsspeicher befinden
3. Es muss von der CPU ausgeführt werden.

Ist eine dieser Bedingungen nicht erfüllt, handelt es sich lediglich um Daten.

„Hä, ich verstehe nur Bahnhof!“ Versuchen wir es anders: Nehmen wir an, ein Programm liegt in einer Programmiersprache vor und wurde in einem Editor erstellt. Dann befinden sich in der Datei lediglich Daten. Diese kann man nämlich einfach mal so editieren. Wenn jetzt dieses Programm vom Compiler in Maschinensprache übersetzt wird, kommt eine neue Datei heraus. Auch in dieser Datei befinden sich nur Daten; sie wurden ja gerade eben vom Compiler erzeugt. Wenn diese Datei nun in den Arbeitsspeicher geladen und von der CPU zur Ausführung gebracht wird, werden die Daten zu einem Programm.

„Können Programme im Arbeitsspeicher aber trotzdem nur Daten und keine Programme sein?“ Klar, wenn der Compiler Maschinencode erzeugt, haben wir lediglich *Daten*, die sich zwischenzeitlich im Arbeitsspeicher befinden, denn sie werden *nicht* von der CPU ausgeführt. *„Ah, ich denke, ich hab's jetzt halbwegs verstanden!“*

Was ist ein Programm? Wie eigentlich schon im vorherigen Punkt angedeutet wurde, ist ein Programm etwas, das in Maschinencode vorliegt. Aber auch die algorithmischen Beschreibungen, die in einer Programmiersprache wie z.B. C vorliegen, werden als Programme bezeichnet, obwohl sie in dieser Form nie zur Ausführung gebracht werden können. Die CPU versteht nämlich kein C sondern nur Maschinencode :-)

Wo befindet sich ein Programm? Wenn ein Programm wirklich ein Programm ist, befindet es sich im Arbeitsspeicher. Das haben wir bereits weiter oben besprochen. Bevor es von der CPU ausgeführt wird, liegt es irgendwo auf den Platten und es handelt sich lediglich um Daten. Unter Linux befinden sich viele der Systemprogramme in den Verzeichnissen `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/usr/local/bin`, `/usr/local/sbin` und `/usr/games`, Die meisten der selbst entwickelten Programme befinden sich in entsprechenden Unterverzeichnissen des eigenen Home-Bereiches.

Wie wird ein Programm gestartet? Viele Laien-Benutzer, insbesondere wenn sie unter Windows arbeiten, starten ein Programm durch Klicken mit der Maus auf entsprechende Bildchen, auch Icons genannt. Vielfach werden Programme auch durch einen Doppel-Klick auf ein Dokument gestartet, dass sich auf dem „Desktop“ befindet. In all diesen Fällen übernimmt die grafische Oberfläche das Starten des richtigen Programms, wovon die meisten Nutzer nichts mitbekommen.

Aber natürlich kann man ein Programm auch durch Eingabe des Programmnamens starten. Dies stammt aus der Zeit, in der man nur einfache Terminals ohne jegliche Grafik mit dem Rechner verbinden konnte. Hinter diesen Terminals lief ein Programm, das im Allgemeinen als Shell, Kommandointerpreter oder auch Kommandozeile (Windows) bezeichnet wird. Dieses Programm nimmt den Programmnamen, sucht die entsprechende Datei auf dem Plattenlaufwerk und bringt es mit Hilfe des Betriebssystems zur Ausführung. Letzteres beinhaltet nach obigen Erläuterungen den Transfer des Maschinen Codes von der Platte zum Arbeitsspeicher und das Überzeugen der CPU davon, dieses Programm auch abzuarbeiten.

Im Rahmen der Lehrveranstaltung werden wir unsere Programme immer mittels Eingabe ihrer Namen starten. Das Schöne daran ist, dass man bei der Eingabe ein Programm mit weiteren Parametern versorgen und somit sein Verhalten steuern kann.

4.5 Zusammenfassung

Ein Rechner besteht aus diversen Hardware-Komponenten, von denen im Rahmen unserer Lehrveranstaltung nur die CPU und das RAM relevant sind. Das Betriebssystem bildet eine Schicht, die die Hardware-Komponenten vor fehlerhaften Zugriffen schützt und somit den Betrieb des Rechners gewährleistet. Der Compiler ist ein Dienstprogramm, das ein in einer Programmiersprache wie C vorliegendes Programm nimmt und in Maschinencode übersetzt. Durch Eingabe des Programmnamens wird dieses Programm mit Hilfe des Betriebssystems in den Arbeitsspeicher transferiert und dort zur Ausführung gebracht. Ausführung bedeutet, dass die CPU *nacheinander* Maschinenbefehl für Maschinenbefehl in die CPU lädt und dort von der Hardware ausführen läßt.

Kapitel 5

Prozessor (CPU) und Arbeitsspeicher (RAM)

Im vorherigen Kapitel haben wir uns angeschaut, aus welchen Teilen ein PC besteht, wie er mittels des Betriebssystems organisiert wird und was es bedeutet, ein Programm zur Ausführung zu bringen. Ferner haben wir darauf hingewiesen, dass im Rahmen unserer Lehrveranstaltung nur die beiden Komponenten CPU und Arbeitsspeicher relevant sind. In diesem Kapitel versuchen wir, einen ersten Eindruck davon zu vermitteln, was diese beiden Teile sind und was sie so machen. In Kapitel 36 greifen wir dieses Thema wieder auf und erklären, was sie mit unseren C-Programmen machen.

5.1 Exkurs: Technik digitaler Systeme

Analog vs. Digital: Früher, zu Zeiten Eurer Großeltern, war alles analog. Selbst in den 80er Jahren gab es noch Analogrechner, die in speziellen Anwendungen ihre Berechtigung hatten. Egal, mit fortschreitender Miniaturisierung der Bauteile fanden Digitalsysteme eine immer größere Verbreitung. Der Unterschied zu Analogsystemen ist, dass in Digitalsystemen Werte nicht mit beliebiger Auflösung sondern in diskreten Schritten kodiert werden. Beispiel: in Analogrechnern kann eine Spannung *jeden beliebigen* Wert zwischen -10 V und +10 V annehmen. In Digitalsystemen sind nur *diskrete* Werte wie beispielsweise ... 4,5 V, 4,6 V, 4,7 V, ... möglich. Die tatsächlichen Werte hängen von der Auflösung ab, die das Digitalsystem zur Verfügung stellt.

Binärwerte: Ein tieferer Blick in unsere Digitalsysteme zeigt, dass alle relevanten Leitungen und Bauteile nur *zwei Werte* annehmen können. Daher spricht man auch von *Binärwerten* oder auch einem *Binärsystem.* Diese beiden Werte werden auch als `an/aus`, `wahr/falsch`, `1/0`, `5 V/0 V` usw. bezeichnet. Alle diese Bezeichnungen sind synonym und können beliebig ausgewechselt werden.

Bits und Bytes: Von einer etwas abstrakteren Perspektive aus betrachtet, können mit

zwei unterschiedlichen Zuständen genau zwei unterschiedliche Werte gespeichert bzw. kodiert werden. Dies nennt man ein `Bit`. Um das Chaos innerhalb eines Digitalrechners nicht unendlich groß werden zu lassen, werden traditionell acht Bits zu einem *Byte* zusammengefasst. Überlicherweise ist das auch die kleinste Einheit innerhalb eines Arbeitsspeichers sowie der CPU. Ferner werden üblicherweise zwei Bytes zu einem *Wort* und vier Bytes (oder zwei Worte) zu einem *Langwort* zusammengefasst. Natürlich hat man innerhalb der CPU genügend Möglichkeiten, auf einzelne Bits zuzugreifen.

Wertekodierung: Mit acht Bits kann man offensichtlich 2^8 verschiedene Werte darstellen (kodieren). Welche Bit-Kombination welchen Wert repräsentiert, hängt von verschiedenen Aspekten ab und ist Sache des CPU-Herstellers, der Programmiersprache sowie des verwendeten Compilers. *„Gibt's einen allgemeingültigen Standard?“* Nein, leider Fehlanzeige. Aber es gibt ein paar Standards, an die sich die meisten Hersteller und Sprach-/Compilerdesigner halten. Die unterschiedlichen Standards rühren vermutlich aus den beschränkten Möglichkeiten früherer Systeme her. Vor 25 Jahren war man megastolz, wenn der eigene Rechner 16-Bit Daten und 32 Kilobytes Arbeitsspeicher hatte. In diesen Zeiten war jedes einzelne Bit wichtig. In den meisten heutigen Systemen ist dies wohl anders, aber die Tradition bleibt bestehen, sodass wir uns damit herumschlagen müssen. An dieser Stelle sei noch angemerkt, dass es in fast allen Anwendungsfällen gar nicht so wichtig ist, wie denn nun die Programmiersprache die einzelnen Werte durch Bits kodiert. Aber manchmal ist es eben doch wichtig, sodass wir dieses Thema nochmals in Skriptteil VIII aufgreifen.

5.2 Der Arbeitsspeicher (RAM)

Während der Programmausführung kommuniziert die CPU im Wesentlichen immer nur mit dem Arbeitsspeicher. Hier befinden sich also alle Programmanweisungen sowie Daten. Selbst wenn sich einzelne Daten in Dateien oder „im Internet“ befinden, werden sie zunächst mit freundlicher Unterstützung des Betriebssystems in den Arbeitsspeicher geladen.

Die kleinste Organisationseinheit innerhalb des Arbeitsspeichers ist *ein Byte*, das in der Regel acht Bits hat. Jedes einzelne Byte hat innerhalb des Arbeitsspeichers eine Adresse. Um ein Byte zu lesen oder zu beschreiben, muss die CPU dem Speicher diese Adresse mitteilen und ihm sagen, ob er lesen oder schreiben möchte. Beides, also Adresse und Zugriffsmodus, teilt die CPU dem Arbeitsspeicher über diverse elektrische Leitungen mit.

Durch die fortschreitende Entwicklung in der Digitaltechnik wurden auch die Anforderungen größer. Daher kann auch ein Arbeitsspeicher zwei, vier oder auch acht Bytes zu einem Wort, Langwort (Double Word) oder Vierfachwort zusammenfassen. Auch hier ist die Terminologie nicht einheitlich, was uns aus Anwendersicht erst einmal nicht weiter stört. Diese Flexibilität erfordert, dass die CPU dem Speicher ebenfalls mitteilt, wie viele Bytes sie gerade zusammengefasst sehen möchte.

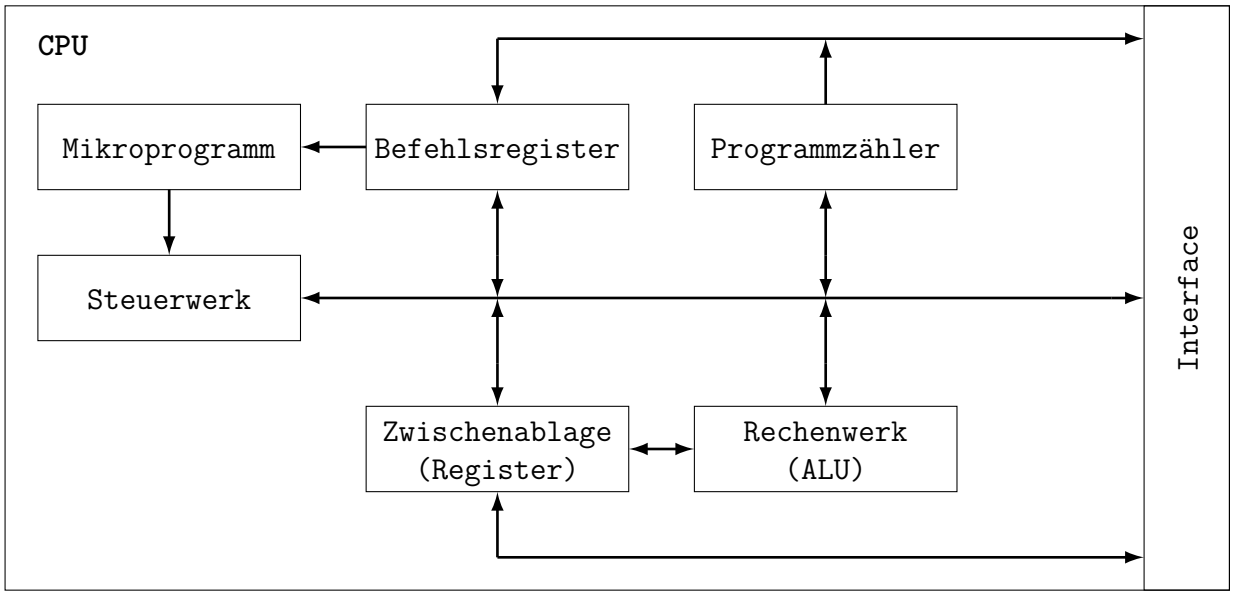
Zum Schluss noch ein paar Zahlen: Ein Arbeitsspeicher mit 4 GB Speicherplatz hat 32 Leitungen, mit denen er in Wirklichkeit $2^{32} = 4.294.967.296$ unterschiedliche Bytes auswählen

kann. Diese 4.294.967.296 Bytes können nach Adam Ries zu 2.147.483.648 Worten zusammengefasst werden. *Well, you get the picture.* Über die kleinen „Rundungsfehler“ bzw. „Nachkommastellen“ machen wir uns erst einmal keine Gedanken. Uns ist es im Allgemeinen egal, ob 1 Kilobyte (KB) nun gemäß Sprachgebrauch genau 1000 oder technisch gesehen 1024 Bytes sind. Sollte es mal wichtig sein, werden wir darauf hinweisen.

Zusammenfassung: Im Arbeitsspeicher gibt es nur einzelne Bits, also Nullen und Einsen, die in Vielfachen von acht Bits organisiert werden. Da der Arbeitsspeicher nur Nullen und Einsen kennt, weiß er auch nicht, was eine einzelne Bit-Kombination bedeuten soll. Ihm ist es also egal und auch unbekannt, ob die Bitkombination `0100 0001` die Zahl 65, der Buchstabe `A` oder ein Additionsbefehl sein soll. Aus Sicht des Arbeitsspeicher ist dies Sache der CPU. Oder anders gesagt: Der Arbeitsspeicher hat keinerlei Informationen über die Bedeutung der einzelnen Werte, die er speichert.

5.3 Der Prozessor

Der Prozessor ist ein echt komplexes Gebilde. Etwas vereinfacht besteht ein Prozessor aus folgenden Komponenten, die die folgenden Aufgaben haben:



Zwischenablage: Die CPU benutzt eine Zwischenablage, um einzelne, aus dem Arbeitsspeicher entnommene Werte oder einzelne Rechenergebnisse temporär abzulegen. Im Vergleich zum Arbeitsspeicher ist diese Zwischenablage sehr dicht am Rechenwerk und arbeitet mit einer wesentlich höheren Taktgeschwindigkeit. Da aber der Realisierungsaufwand recht hoch ist, können hier nur wenige Werte abgelegt werden.

Rechenwerk: Das Rechenwerk führt alle Rechenoperationen aus. Dazu zählen alle arithmetischen sowie logischen Operationen. Dieses Rechenwerk ist insbesondere auf Ge-

schwindigkeit ausgelegt, damit der Prozessor möglichst schnell getaktet werden kann. Aufgrund dieser Funktionalität wird das Rechenwerk auch als Arithmetic-Logical Unit (ALU) bezeichnet.

Befehlsregister: Hier befindet sich die momentan abzuarbeitende Maschineninstruktion. Diese Instruktionen werden auch als OpCodes bezeichnet und sind vor allem Rechenoperationen sowie Datentransporte von und zum Arbeitsspeicher. Diese OpCodes können fast jede Bit-Kombination annehmen und sind somit *nicht* von den Werten (Operanden) unterscheidbar. Anders ausgedrückt: anhand der Bit-Kombination kann man nicht unterscheiden, ob es sich um eine Maschineninstruktion oder einen Operanden handelt. Die Bedeutung der Bit-Kombination muss die CPU „wissen“, sie muss aus der Programmstruktur hervorgehen.

Befehlszähler: Hier steht die Adresse, unter der sich die nächste Maschineninstruktion im Arbeitsspeicher befindet. Anders ausgedrückt: Hier ist abgelegt, wo im Arbeitsspeicher der nächste Befehl des gerade aktiven Programms zu finden ist.

Mikroprogramm: Im Mikroprogramm steht (fest verdrahtet), wie eine Maschineninstruktion im Einzelnen abzuarbeiten ist. Hier ist also festgelegt, ob noch weitere Operanden aus dem Arbeitsspeicher nachgeladen werden müssen, welche Register aus der Zwischenablage verwendet werden sollen und welche Rechenoperation das Rechenwerk ausführen soll.

Steuerwerk: Das Steuerwerk bildet zusammen mit dem Mikroprogramm das „Master Brain“ der CPU. Es koordiniert das Zusammenspiel aller einzelnen Komponenten und darüberhinaus auch über das Interface die Aktivitäten der weiteren angeschlossenen Baugruppen.

Generelle Funktionsweise: Ein Programm wird nun wie folgt abgearbeitet: Der Befehlszähler zeigt auf die nächste, abzuarbeitende Instruktion. Diese wird durch das Steuerwerk aus dem Arbeitsspeicher ausgelesen und in das Befehlsregister abgelegt. Von dort geht es zum Mikroprogramm, das diese Instruktion in kleine Schritte zerlegt und diese auf Hardware-Ebene ausführt. Parallel dazu wird der Inhalt des Befehlszählers erhöht, sodass er auf die nächste Instruktion zeigt.

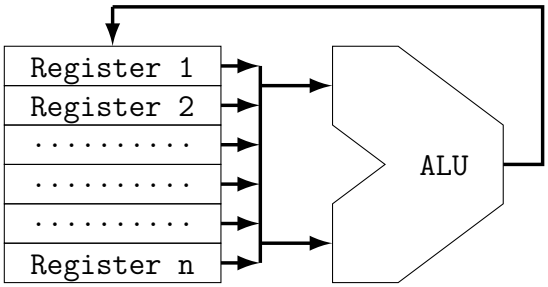
Die Abarbeitung einer Instruktion kann weitere Operanden benötigen, die ggf. aus dem Arbeitsspeicher nachgeladen werden. Das (Nach-) Laden der Operanden führt ebenfalls zu einer Erhöhung des Befehlszählers, damit dieser nicht auf die Operanden, sondern auf die nächste Instruktion zeigt.

Auch die CPU weiß nichts über das Programm, das sie gerade abarbeitet. Die CPU nimmt Instruktion für Instruktion und führt diese ohne weiteres Nachdenken aus. Dies klappt nur, solange die Struktur, also die entsprechenden Bit-Kombinationen, im Arbeitsspeicher korrekt sind. Die CPU kann hier nichts korrigieren.

Zwischenablage und Rechenwerk: Das Rechenwerk ist eng an die Zwischenablage gekoppelt, denn von hier werden die Operanden bezogen und die Ergebnisse hinein geschrie-

ben. Eine Zwischenablage besteht je nach Prozessor aus ungefähr 32 bis 1024 einzelnen Registern. Eine etwas detailliertere Darstellung dieser beiden Komponenten sieht wie folgt aus:

Diese Grafik zeigt, dass, durch das Steuerwerk kontrolliert, die ALU die Operanden aus der Zwischenablage (Register File) auswählt, diese verarbeitet und das Ergebnis wieder in eines der Register zurückschreibt.



5.4 Programmierung

Bis hier her sollte unter anderem folgendes klar geworden sein: Um ein richtiges Programm zu erstellen, muss man die richtigen Nullen und Einsen in den Arbeitsspeicher bekommen. Selbst wenn man diese Nullen und Einsen hat, benötigt man ein wenig Hardware- und Software-Unterstützung, die im Allgemeinen gegeben ist.

Direkte Nullen und Einsen: Für ein einfaches Programm, dass lediglich die Nachricht `Hi, I am born` ausgibt, sieht das ungefähr so aus:

```
  1 0000000 042577 043114 000401 000001 000000 000000 000000 000000
  2 0000020 000002 000003 000001 000000 101460 004004 000064 000000
  3 0000040 010514 000000 000000 000000 000064 000040 000010 000050
  4 0000060 000036 000033 000006 000000 000064 000000 100064 004004
  5 0000100 100064 004004 000400 000000 000400 000000 000005 000000
  6 0000120 000004 000000 000003 000000 000464 000000 100464 004004
  7 0000140 100464 004004 000023 000000 000023 000000 000004 000000
  8 0000160 000001 000000 000001 000000 000000 000000 100000 004004
    ....... ...... ...... ...... ...... ...... ...... ...... ......
288 0016000 057400 067151 072151 000000
```

Nur haben wir hier zwei Dinge drastisch vereinfacht: Wir haben alle Nullen und Einsen in Gruppen von drei Ziffern zu einer (Oktal-) Zahl zusammengefasst. Ferner haben wir nach ein paar Zeilen abgebrochen, denn der komplette Ausdruck würde sich über knapp fünf Seiten erstrecken. Wer dazu Lust hat, bitte schön. Der Erfinder des freiprogrammierbaren Rechners, Konrad Zuse, musste dies während des 2. Weltkrieges tatsächlich so machen. Wir machen es nicht so, denn selbst am Ende des Semesters würden wir über so eine einfache Ausgabe nicht hinaus kommen.

Assemblerprogramierung: Eine Ebene über den Nullen und Einsen gibt es eine sehr primitive Programmiersprache, die prozessorspezifisch ist und Assembler genannt wird. Für unser Beispiel sieht das Assembler-Programm wie folgt aus:

```
1           .file   "hello.c"
2           .section        .rodata
3   .LC0:
4           .string "Hi, I am born"
5           .text
6   .globl main
7           .type   main, @function
8   main:
9           pushl   %ebp
10          movl    %esp, %ebp
11          andl    $-16, %esp
12          subl    $16, %esp
13          movl    $.LC0, (%esp)
14          call    puts
15          leave
16          ret
17          .size   main, .-main
18          .ident  "GCC: (Ubuntu 4.4.3-4ubuntu5.1) 4.4.3"
19          .section        .note.GNU-stack,"",@progbits
```

Hier sieht man schon 'was, doch ist das Programm alles andere als verständlich. Bei brandneuer Hardware muss man manchmal so programmieren. Aber die Erstellung einigermaßen sinnvoller Programme ist mehr als mühsam. Das machen wir also auch nicht. Aber bitte, wer möchte, der kann ;-)

C-Programmierung: In der Programmiersprache C sieht das Programm wie folgt aus:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4      {
5          printf( "Hi, I am born\n" );
6      }
```

Schon besser. Ist zwar immer noch nicht ganz einfach, aber wenn wir in C programmieren, können wir am Ende des Semesters doch schon recht komplexe Programme erstellen. Um so ein Programm zu Laufen zu bringen, müssen wir den Quelltext in eine Datei, beispielsweise `hello.c` schreiben, dann mittels des Compilers `gcc -o hello hello.c` in Maschinencode übersetzen und unter Linux mittels `./hello` zur Ausführung bringen. Diese Schritte schauen wir uns in Kapitel 39 genauer an.

Und genau um diese Form der C-Programmierung geht es das ganze Semester lang.

Kapitel 6

Software Life Cycle: Von der Idee zum Programm

Der Begriff *Software Life Cycle* beschreibt den Weg eines Programms von der Idee bis zu seiner Realisierung auf dem Rechner und die später notwendigen Wartungsarbeiten. Hierfür gibt es in der Literatur sehr viele Modelle. Das hier beschriebene Modell teilt den Software Life Cycle in folgende Phasen ein: Aufgabenstellung, Problemanalyse, Entwurf, Implementierung und Test. Anschließend folgt noch die Codierung in der gewählten Programmiersprache.

Die Aufteilung in die einzelnen Phasen und die Durchführung der darin zu erledigenden Arbeitsschritte sind weder Gesetze noch müssen sie akurat abgearbeitet werden. Vielmehr handelt es sich um Handlungsrichtlinien bzw. Empfehlungen. Der Programmieranfänger tut aber gut daran, sich *in etwa* daran zu halten, sonst sind Misserfolg und Frust unausweichlich. Und das wollen wir ja nicht ...

6.1 Motivation

Mit dem Programmieren ist es wie mit dem Bearbeiten anderer größerer Aufgaben, sie sind komplex, man weiss nicht so recht, was man tun soll, findet nicht so recht den Anfang und sieht anfänglich den Wald vor lauter Bäumen nicht. Hier helfen nur Erfahrung und Systematik, aber beides hat man als Anfänger nicht.

Aufgrund der mangelnden Erfahrung, aber dem Willen, etwas zu tun, wird meistens sehr schnell drauf los getippt. *Etwas mal eben probieren* ist etwas, das wir explizit unterstützen! Aber im Hinblick auf das Lösen einer gestellten Programmieraufgabe führt das meist in die falsche Richtung, zu Misserfolg, zu Frust.

Daher raten wir als Dozenten jedem von Euch, sich zumindest bei den ersten Aufgaben in etwa an die einzelnen Phasen des Software Life Cycles zu halten. Ein systematisches,

schrittweises Vorgehen sieht in etwa wie folgt aus: herausfinden, was die Aufgabe eigentlich von einem will, was die Ein- und Ausgaben sein sollen, wie sich das Programm verhalten soll (Problemanalyse), die Aufgabe in kleine, zu bewältigende Teilaufgaben unterteilen (Entwurf), die einzelnen Funktionalitäten schrittweise immer genauer beschreiben, bis man nur noch ganz einfache, simple Anweisungen übrig hat (Implementierung) und zum Schluss das Eintippen (Kodieren) und Testen. Zudem sei gesagt: Je früher man einen Fehler im Gesamtprozess findet, um so leichter ist es, ihn zu beheben.

Auch wenn es vielleicht etwas schwer fällt, das alles zu glauben, so hilft es doch. Die nächsten Abschnitte stellen kurz *ein* mögliches Phasenmodel des Software Life Cycles vor. Da unsere Absolventen später nicht primär Software Engineering betreiben, sondern mittels kleinerer Testprogramme ihre selbst entwickelten (Hardware-) Systeme testen und validieren wollen, erfolgt die Beschreibung relativ informell.

Die einzelnen Phasen werden mittels zweier kleiner Beispiele illustriert. Das erste Beispiel ist ein Alltagsproblem, nämlich der Bau eines Hauses, das zweite ein erstes kleines Programmierproblem, die Berechnung der Fläche eines Rechtecks.

6.2 Aufgabenstellung: worum geht's?

„Es werde Licht." So oder so ähnlich fängt alles an. Am Anfang steht meistens eine Aufgabenstellung oder ein Kundenauftrag, die bzw. der vom Auftragnehmer umgesetzt werden muss. Ein Charakteristikum einer derartigen Aufgabenstellung ist, dass sie zwar kurz und verständlich, aber dennoch recht vage ist. Zwei Beispiele könnten wie folgt aussehen:

`Hausbau`

Eine vierköpfige Familie benötigt ein eigenes Haus, da ihre Einzimmerwohnung zu klein geworden ist.

`Fläche eines Rechtecks`

Eine Studentin will die Grundfläche ihrer rechteckig geschnittenen Wohnung berechnen.

6.3 Problemanalyse: das *Was*?

Diese Phase dient vor allem dazu, sich Klarheit über die Aufgabenstellung zu verschaffen. Am Ende dieser Phase steht ein *Pflichtenheft,* mit dessen Hilfe beide Seiten, also Auftraggeber und Auftragnehmer, einverstanden sind. Unter anderem sollten folgende Fragen beantwortet werden:

1. Was sind die Ein- und Ausgaben, welche Werte sind sinnvoll bzw. gültig?
2. Wie wird die Bedienung gestaltet, was soll im Fehlerfalle passieren?
3. Welche Formeln und welche Einheiten werden wo benötigt und verwendet?

Dies sind nur ein paar Beispiele. Diese zeigen auch, dass sich die einzelnen Fragestellungen damit beschäftigen, *was gemacht werden soll* und *nicht* damit, *wie* es realisiert wird.

Entsprechend sind alle Arbeiten von der Wahl einer konkreten Programmiersprache unabhängig.

Das Pflichtenheft erfüllt unter anderem zwei wesentliche Aufgaben. Erstens hilft es beiden Seiten, sich zu verständigen, denn in der Regel sprechen sie unterschiedliche Fachsprachen. Zweitens ist das Pflichtenheft für beide Seiten *verbindlich* (daher auch der Name): Im Problemfalle wird mit dessen Hilfe beurteilt, ob ein Programm korrekt ist oder nicht. Die Pflichtenhefte unserer beiden Beispiele könnten wie folgt aussehen:

`Hausbau`

Für den Bau eines Hauses für die vierköpfige Familie könnten folgende Fragen relevant sein: Gibt es ein gemeinsames Wohnzimmer? Wie groß sollte jedes Zimmer sein? Soll sich die Wohnung über eine oder vielleicht über zwei Etagen erstrecken? Wie groß sollte die Raumhöhe sein? Wie viel Zugänge werden benötigt?

`Fläche eines Rechtecks`

Folgende Punkte könnten definiert werden: Die Fläche F eines Rechtecks mit den Kantenlängen a und b beträgt $F = a \times b$. Sinnvolle Werte für a und b sind positive Zahlen; Längen kleiner gleich null sind unsinnig. Alle Längenangaben werden in m getätigt. Die Ausgabe besteht aus einer kleinen Meldung und dem Flächeninhalt.

6.4 Entwurf: *Welche* Funktionseinheiten?

Im Gegensatz zu unseren beiden didaktisch orientierten Minibeispielen sind konkrete, praxisrelevante Aufgabenstellungen meist sehr komplex. Daher besteht der erste Schritt meist in einer ersten Aufteilung des Ganzen in mehrere möglichst eigenständige Komponenten, die anschließend möglichst unabhängig voneinander (von unterschiedlichen Teams) weiter bearbeitet werden können. Diese Phase nennt man auch *Programmieren im Großen*, deren Ergebnis eine erste Modulstruktur ist, die die benötigten Funktionalitäten durch einzelne Module abbildet. Erst in der nächsten Phase werden diese Module implementiert. In unseren beiden Beispielen könnten wir wie folgt herangehen:

`Hausbau`

Hier könnten wir klären, was für Wände, Fenster und Türen benötigt werden, ob ein Kamin eingebaut werden soll und ob Platz für eine Terasse benötigt wird.

`Fläche eines Rechtecks`

Wir brauchen folgende Funktionalitäten:

- Einlesen der beiden Kantenlängen
- Ausgabe von Fehlermeldungen
- Berechnung der Fläche
- Ausgabe von Ergebnissen

Aufgrund dessen, dass Ihr Programmieranfänger seid, sind unsere ersten Programmieraufgaben sehr klein und übersichtlich. Daher werden in der ersten Hälfte des Semesters die Entwurfsphasen eher rudimentär ausfallen. Dennoch werden wir versuchen, die grundlegenden Ideen dieser Phase von Anfang an zu vermitteln.

6.5 Implementierung: das *Wie*?

Langsam wird es ernst, denn es geht um das *wie*! Diese Phase nennt man auch *Programmieren im Kleinen.* Jetzt geht es darum, die in der vorherigen Phase identifizierten Funktionalitäten konkret zu implementieren (realisieren). Im Gegensatz zur landläufigen Meinung bedeutet „Implementieren“ aber nicht das Eintippen von C-Code. Diese als *Coding* bezeichnete Tätigkeit ist Gegenstand der nächsten Phase.

In der Phase der Implementierung werden für die einzelnen Funktionalitäten algorithmische Beschreibungen entwickelt. Zur Strukturierung dieses Prozesses bietet die Literatur eine Reihe von Methoden an. Im Rahmen der Vorlesung verwenden wir *Struktogramme* und die *Methode der (verbalen) Schrittweisen Verfeinerung,* die beide ausführlicher im nächsten Skriptteil vorgestellt werden. Für den Laien ohne Programmierkenntnisse lässt sich die zweite Methode an folgendem Beispiel erklären:

```
Kaffee kochen
Richtige Menge Wasser einfüllen
Filter einlegen
Richtige Menge Kaffeepulver in Filter einfüllen
Kanne unter den Filter stellen
Kaffeemaschine anschalten
```

Wie man sieht, besteht ein derartiger Algorithmus aus der Aneinanderreihung der durchzuführenden Tätigkeiten, wobei jeder einzelne Arbeitsschritt nach Bedarf beliebig weiter verfeinert werden kann (daher auch der Name *Schrittweise Verfeinerung*). Beispiel:

```
Richtige Menge Kaffeepulver in Filter einfüllen
Dose aus dem Schrank holen
Dose öffnen
Löffel aus der Schublade holen
Pro Tasse 2 Löffel Pulver in den Filter streuen
Dose schließen
Dose in den Schrank zurückstellen
Löffel in die Schublade zurücklegen
```

Software Algorithmen kann man auf die gleiche Art und Weise entwickeln. Nur muss man hier die notwendigen Datenstrukturen (Variablen) und Anweisungen beschreiben. Auf das Problem der Flächenberechnung angewendet, könnte dies zu folgenden Algorithmen führen:

```
Fläche eines Rechtecks
Variablen: Integer: a, b, F
Einlesen der Seite a
Einlesen der Seite b
Berechne Fläche F = a * b
Ausgabe des Flächeninhaltes F
```

```
Einlesen der Seite x
Ausgabe des Textes:
Bitte Wert für Seite x eingeben
Lese Wert für Seite x
```

```
Ausgabe des Flächeninhaltes F
Ausgabe des Textes: Der Flächeninhalt beträgt:
Ausgabe des Wertes von F
```

Das war's dann auch schon :-), so einfach geht's. Das Beispiel zeigt sehr schön, wie einzelne (komplexe) Anweisungen, beispielsweise `Einlesen einer Seite` und `Ausgabe der Fläche` in einem weiteren Schritt (rechter bzw. unterer Teilalgorithmus) verfeinert werden können.

Zum Abschluss nochmals die Vorteile, die sich aus dem Verwenden der beiden angesprochenen Methoden, Struktogramme und Schrittweise Verfeinerung, ergeben (können):

1. Beide Methoden bieten eine Strukturierungshilfe, um die Gedanken und damit die zu entwickelnden Algorithmen zu strukturieren. Sonst kommt es oft zu sogenanntem Spaghetti-Code, den keiner versteht und der voll von Fehlern ist.
2. Die Trennung von (abstrakter) Implementierung und (konkreter) Kodierung reduziert die Komplexität des gesamten Prozesses: Erst konzentriert man sich auf die Funktionsweise des Algorithmus ohne jeglichen Programmiersprachen-Schnickschnack und dann kümmert man sich um die Programmiersprachendetails ohne sich noch Gedanken um den Ablauf machen zu müssen. In der Folge kommt man wesentlich schneller zu einem korrekt funktionierenden Programm.
3. Man spart sehr viel Zeit, wenn man diesen Arbeitsschritt mit Papier und Bleistift durchführt.

6.6 Kodierung: Eintippen und Übersetzen

Wie in der vorherigen Phase beschrieben, geht es in dieser Phase um die Umsetzung des entwickelten Algorithmus in eine konkrete Programmiersprache. Diese Tätigkeit kann in der Regel recht gradlinig, nahezu maschinell erfolgen. Den ersten Teil des notwendigen Rüstzeugs hierfür präsentieren wir in Skriptteil III.

6.7 Test: funktioniert alles wie gewünscht?

Testen ist nicht eine lästige Phase, die am Ende noch eben gemacht werden muss, sondern eine Tätigkeit, die den gesamten Prozess begleiten sollte. Insbesondere im Rahmen der Lehrveranstaltung sollten alle Ergebnisse ab der Problemanalyse in Form einer Handsimulation getestet werden. Dazu nimmt man sich Blatt und Papier und spielt die einzelnen Anweisungen durch. Je später man einen Fehler findet, um so schwieriger wird es, ihn wieder zu beseitigen.

Um sinnvoll testen zu können, sollte man sich ab der Problemanalyse Testdaten zurechtlegen. Um die einzelnen Entwürfe und später auch die Programme zu validieren, ist eine gewisse Systematik sinnvoll. Dazu sollten die Testdaten folgende Fälle berücksichtigen:

1. Gültige Eingaben die repräsentativ für den Normalfall sind.
2. Datensätze, bei denen einzelne Werte gerade noch gültig oder gerade eben ungültig sind.
3. Weitere Datensätze, damit jeder Programmzweig mindestens einmal durchlaufen wird.

Für unsere Flächenberechnung eines Rechtecks bieten sich beispielsweise folgende Eingabewerte an:

Fläche eines Rechtecks:

`a`	10	10	0	-4	-1
`b`	4	40	13	10	-1
`F`	40	400	0	-40	1

Zur Erinnerung: In unserem Beispiel der Flächenberechnung eines Rechtecks werden zwei Seiten a und b eingelesen, die gemäß Problemanalyse nur positive Werte haben sollen. Insofern sind die ersten beiden Datensätze aus dem gültigen Wertebereich, wohingegen die anderen aus dem ungültigen sind.

Kapitel 7

Mein erstes C-Programm: Fläche eines Rechtecks

In Kapitel 6 haben wir die Flächenberechnung eines Rechtecks von der Aufgabenstellung bis hin zur (virtuellen) Implementierung durchgeführt. Das Ergebnis war eine recht feingliedrige Beschreibung, die recht dicht an den Möglichkeiten fast aller Programmiersprachen ist. Dieses Kapitel zeigt die konkrete Umsetzung in die Programmiersprache C, beschreibt die einzelnen Programmelemente und erläutert kurz, wie wir den Prozessor dazu bringen, dieses C-Programm auch auszuführen.

7.1 Das C-Programm

Ein entsprechendes C-Programm könnte wie folgt aussehen:

```
1  #include <stdio.h>
2
3  int main( int argc, char ** argv )
4      {
5          int a, b, F;
6          printf( "Bitte Seite a eingeben: " );
7          scanf( "%d", & a );
8          printf( "Bitte Seite b eingeben: " );
9          scanf( "%d", & b );
10         F = a * b;
11         printf( "Der Flaecheninhalt betraegt F=%d m*m\n", F );
12     }
```

Regel Nummer eins: Don't panic! Wir gehen jetzt das Programm Schritt für Schritt durch. Bei dieser ersten Besprechnung werden viele Dinge etwas unklar bleiben. Das ist ganz normal. Aber irgendwo muss man anfangen und daher muss man erst einmal hinnehmen,

dass gewisse Dinge so im Programm stehen *müssen,* wie sie nun mal da stehen. Wir werden im Laufe des Semester auf alle Einzelheiten genau eingehen, versprochen. Und dann werden viele wünschen, dass wir es doch lieber im Dunklen gelassen hätten ...

Kurzüberblick: Wenn wir den Blick einmal kurz über das Programm streifen lassen, sieht man ziemlich klar, dass sich die einzelnen Anweisungen zwischen den Zeilen 5 und 11 befinden. Offensichtlich wird in Zeile 5 dem Compiler klar gemacht, dass es sich bei `a`, `b` und `F` um Variablen für unsere Werte handelt. Und die Anweisungen[1] `printf( ... )` und `scanf( ... )` sind offenkundig für unsere Aus- bzw. Eingaben zuständig. Das ist eigentlich auch schon mal alles. Im Folgenden besprechen wir jede Zeile im Detail

Zeile 1:
Diese Zeile benötigen wir, damit wir etwas auf den Bildschirm ausgeben und von der Tastatur einlesen können. Die Datei `stdio.h` befindet sich irgendwo auf dem Rechner und beschreibt, wie die Ein- und Ausgabeanweisungen konkret aussehen. Durch die spitzen Klammern `<>` weiß der Compiler, dass es sich bei der Datei `stdio.h` um eine Standardbibliothek handelt, deren Ort er bereits durch seine Installation auf dem Rechner weiß. Für die Neugierigen: Unter Linux ist dies die Directory `/usr/include`. Die Direktive `#include` veranlasst den Compiler, die angegebene Datei `stdio.h` in den Quelltext einzubinden.

Zeile 3:
Hier fängt das (Haupt-) Programm an, das in der Programmiersprache C immer `main` heißen *muss*. Nach dem Laden durch das Betriebssystem fängt die CPU ihre Arbeit immer hier an[2].

Die Angaben `int` vor `main` und `(int argc, char **argv)` nach `main` müssen einfach da stehen. Deren Bedeutung besprechen wir in Kapitel 51.

Zeile 4 und 12:
Die geschweiften Klammern begrenzen den Anfang und das Ende des auszuführenden Anweisungsteils; auch diese Klammern *müssen* einfach sein.

Zeile 5-11:
Hier ist die eigentliche Action, hier befinden sich alle auszuführenden Anweisungen.

Zeile 5:
Hiermit wird dem Compiler gesagt, dass wir drei Variablen benötigen, die die Namen `a`, `b` und `F` haben. Das vorangestellte `int` besagt, dass diese Variablen als Werte immer nur ganze Zahlen annehmen können.

Zeile 6 und 8:
Das `printf( ... );` ist eine typische Ausgabeanweisung in C. Es nimmt das Argu-

[1] Bei `main()`, `printf()` und `scanf()` handelt es sich eigentlich nicht um Anweisungen sondern um Funktionen. Aber das würde jetzt zu sehr ablenken, daher mehr dazu in Kapitel 44.

[2] Davor kommt noch ein bisschen was, aber das besprechen wir erst in Kapitel 40, insbesondere Abschnitt 40.3, denn für den Moment ist dies unwichtig und würde uns nur ablenken.

ment `"Bitte Seite a eingeben: "` bzw. `"Bitte Seite b eingeben: "` und gibt es auf dem Bildschirm aus.

Zeile 7 und 9:
Bei `scanf( ... );` handelt es sich um eine typische Eingabeanweisung. Das Argument `"%d"` besagt, dass eine ganzzahlige Zahl erwartet wird. Daran muss man sich dann als Nutzer auch halten, sonst geht's schief.

Das zweite Argument `& a` sagt der Eingabeanweisung, wohin die eingelesene Zahl im Arbeitsspeicher geschrieben werden soll; das `&`-Zeichen (auch Adressoperator genannt) bestimmt die Adresse der nachfolgenden Variablen im Arbeitsspeicher. Genau dort hin schreibt `scanf()` die eingelesene Zahl, sodass wir sie anschließend auch verwenden können.

Zeile 10:
Diese Zeile berechnet einfach das Ergebnis.

Zeile 11:
Wie bereits klar sein sollte, handelt es sich hier wieder um eine Ausgabeanweisung. Im Unterschied zu den Zeilen 6 und 8 hat hier die Ausgabeanweisung `printf()` einen zweiten Parameter `F`. Ferner fällt auf, dass sich im ersten Argument, `"Der Flaecheninhalt betraegt F=%d m*m\n"`, wie bereits bei den beiden `scanf()` Anweisungen, ein `%d` befindet. Wie zuvor ist das der Hinweis auf ein ganzzahliges Argument. Beim Ausgeben ersetzt das `printf()` diese beiden Zeichen durch den *Wert* der übergebenen Variablen. Bei Eingabe von `4` und `12` würde folgende Ausgabe auf dem Bildschirm erscheinen[3]: `Der Flaecheninhalt betraegt F=48 m*m`. Die Zeichenfolge `\n` sorgt dafür, dass *nach* der Ausgabe der Cursor (Schreibmarke) auf den Anfang der nächten Zeile gesetzt wird.

Zeile 5-11: Jede einzelne Anweisung muss mit einem Semikolon abgeschlossen werden.

So, geschafft ...

[3]Die beiden Anführungsstriche `"` werden nicht ausgegeben; sie sind nur dazu da, dem Compiler zu sagen, dass hier Text und keine Variablen oder Anweisungen kommen.

Kapitel 8

Eintippen, Übersetzen und Starten eines Programms

Nach dem letzten Kapitel sind wir nun so weit, dass wir ein C-Programm auf dem Blatt Papier haben. Jetzt müssen wir uns wieder an das erinnern, was wir in Kapitel 5 besprochen haben: Die CPU ist nur in der Lage, Maschinenbefehle auszuführen; sie weiß nichts von C-Programmen, Mathlab, HTML, Datenbankabfragen oder dergleichen. Also müssen wir uns in diesem Kapitel noch darum kümmern, wie wir unser C-Programm in die CPU bekommen. Dies erfordert die folgenden drei Arbeitsschritte:

1. Eintippen des Programms und Abspeichern, z.B. unter dem Namen `bsp.c`[1].
2. Übersetzen des C-Programms in ein entsprechendes Maschinenprogramm mittels des Compilers. Das Ergebnis könnte dann in der Datei `bsp` (Linux) oder `bsp.exe` (Windows) liegen.
3. Ausführen des Programms. Sollte alles funktionieren, Sektkorken knallen lassen.

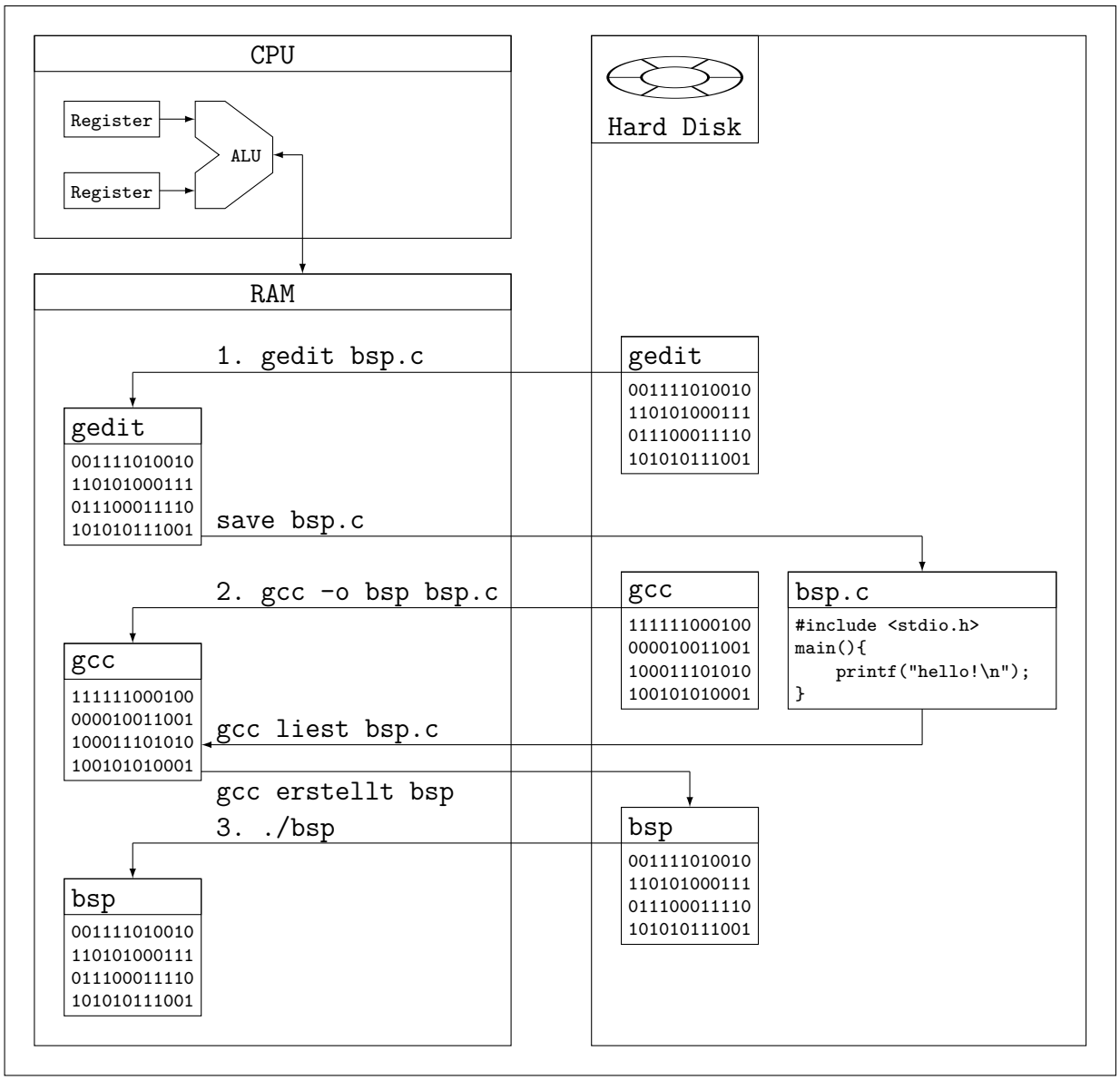
Diese drei Arbeitsschritte und vor allem deren Verbindung miteinander ist wieder recht komplex. Daher haben wir versucht, diese Zusammenhänge auf der nächsten Seite mittels eines kleinen Bildchens zu visualisieren.

8.1 Eintippen des Programms

Um ein Programm in den Rechner zu bekommen, benötigen wir einen Editor. Dieser ermöglicht es uns, alles einzutippen, zu korrigieren, zu löschen usw. Anschließend müssen wir das Getippte noch abspeichern, denn sonst ist alles wieder weg. Für diese Tätigkeit können wir unter Linux beispielsweise das Kommando `gedit bsp.c` verwenden. Unter

[1] Aus Platzgründen haben wir die Namen in den Bildern sehr kurz halten müssen. Aus Gründen der Konsistenz behalten wir die Namen weitestgehend bei.

Erstellen, Übersetzen und Ausführen des Programms bsp bzw. bsp.c



Programme		**Datendateien**	
Editor	: gedit	Editor	: gedit
Compiler	: gcc	Compiler	: gcc
Programm	: bsp	C-Programm	: bsp.c
		Programm	: bsp

Windows wäre beispielsweise `notepad bsp.c` möglich. Am Ende haben wir eine Datei `bsp.c`, in der sich unser C-Programm auf dem Plattenlaufwerk (Hard Disk) unseres Rechners befindet, das eben noch auf dem Blatt Papier stand.

Hinweis: Die Betriebssysteme (eigentlich die Compiler) gehen davon aus, dass sich ein C-Programm in einer Datei befindet, die mit `.c` endet.

8.2 Übersetzen des Programms

Unser C-Programm liegt ja immer noch nicht in Maschinen-Code sondern in normalen druckbaren Zeichen vor. Für diese Umwandlung benötigen wir den C-Compiler[2] `gcc`, der für uns den Maschinen-Code generiert, sofern wir keinen Tippfehler eingebaut haben. Je nach Betriebssystem geht dies wie folgt:

	Linux	Windows
	`gcc bsp.c`	`gcc bsp.c`
oder:	`gcc -o bsp bsp.c`	`gcc -o bsp.exe bsp.c`

Im ersten Fall wird das Ergebnis in die Datei `a.out` (Linux) bzw. `a.exe` (Windows) geschrieben. Im zweiten Fall erscheint das Ergebnis in der Datei, deren Namen wir hinter der Option `-o` angegeben haben.

8.3 Starten des Programms

Je nach Betriebssytem und Option können wir unser Programm wie folgt starten:

	Linux	Windows
`gcc bsp.c`:	`./a.out`	`a.exe`
`gcc -o bsp bsp.c` :	`./bsp`	`bsp.exe`

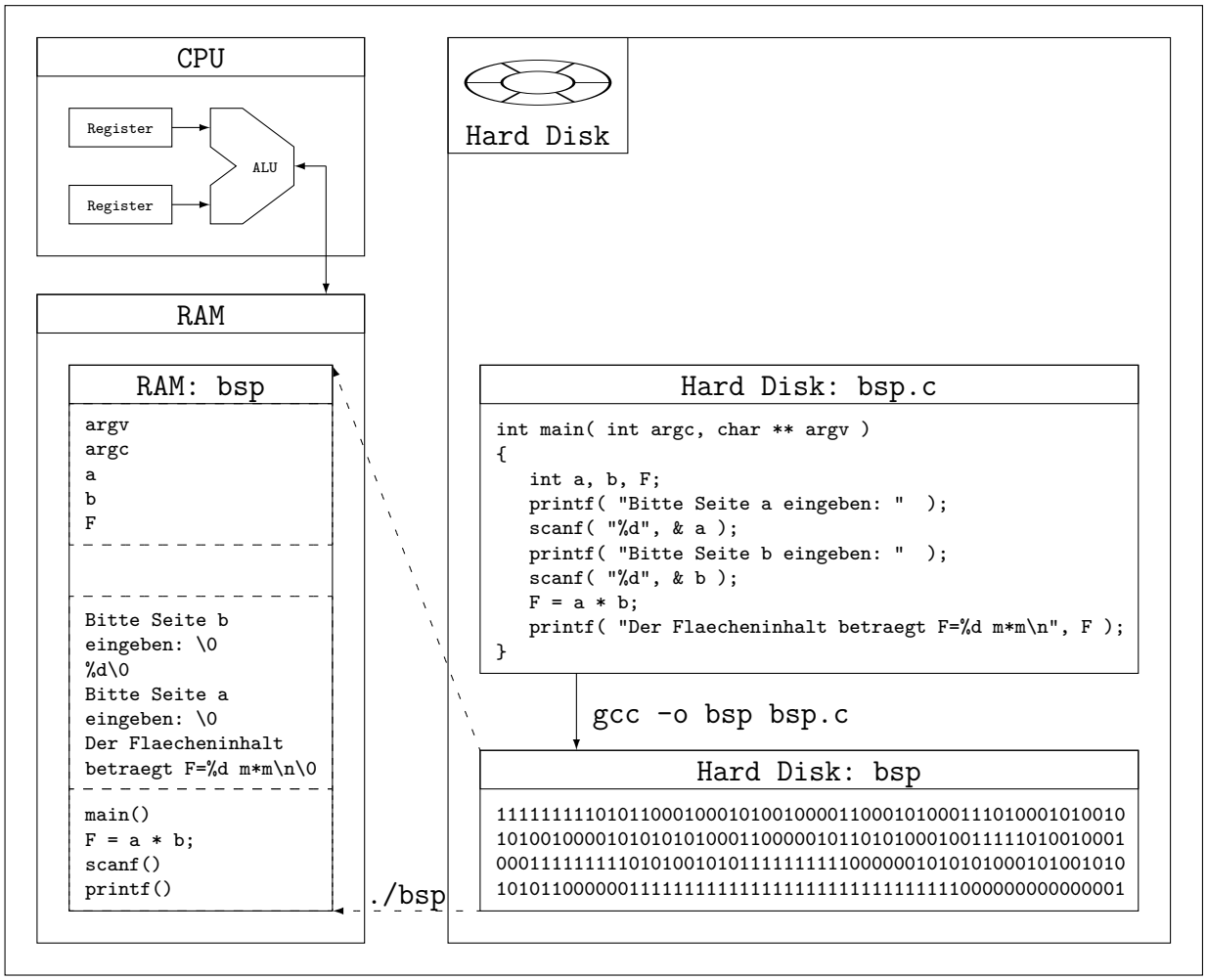
8.4 Was geht eigentlich im Arbeitsspeicher vor?

„Ok, verstanden habe ich, dass die CPU nur Maschinen-Code ausführen kann, der nebenbei bemerkt für uns völlig unverständlich ist. Aber irgendwo müssen doch unsere Variablen und Anweisungen geblieben sein?“ Völlig richtig! Von irgendwo her muss `printf()` ja den Wert der Variablen `F` holen und irgendwo muss `scanf()` ja die eingegebenen Werte für `a` und `b` ja auch hin schreiben. Die Frage ist jetzt nur, wo sie sind und wie ich das heraus bekomme.

Dazu haben wir auf dem Bild der vorherigen Seite etwas genauer in den Arbeitsspeicher geschaut. Auf der rechten Seite sehen wir die Datei `bsp.c`, die mittels des Compilers `gcc`

[2] Je nach Betriebssystem und oder Installation, kann der Name des Compilers auch anders lauten.

Erstellen, Übersetzen und Ausführen des Programms bsp bzw. bsp.c



`-o bsp bsp.c` in Maschinen-Code umgewandelt und in der Datei `bsp` abgelegt wurde. Beide Dateien befinden sich auf der Hard Disk.

Auf der linken Seite sehen wir die Konfiguration im Arbeitsspeicher. Bei genauem Hinsehen fällt folgendes auf:

1. Im Arbeitsspeicher scheint das Programm `bsp` aus drei unterschiedlichen Segmenten zu bestehen.
2. Es scheint so, als habe der Compiler alle Anweisungen in das untere Segment, alle Ausgabetexte, in das mittlere und alle Variablen in das obere Segment gepackt. Richtig erkannt!

Und da wir neugierig sind, werden wir uns vom Programm alle Adressen ausgeben lassen. Dazu haben wir folgende Möglichkeiten:

Adresse einer Variablen: Die Adresse einer Variablen erhält man einfach mittels eines vorangestellten `&`-Zeichens. Für die Variable `a` wäre das beispielsweise: `& a`. Diese Form ist eigentlich nichts Neues, denn wir haben sie bereits bei der `scanf()`-Anweisung gesehen.

Größe einer Variablen: Die Zahl der Bytes, die eine Variable belegt, gibt uns die Anweisung `sizeof()`. Für die Variable `a` wäre dies: `sizeof( a )`

Adresse einer Funktion: Die Adresse einer Funktion erhalten wir in gleicher Weise wie die einer Variablen. Für die Funktion `main()` wäre das einfach: `& main`[3].

Ausgabe der Adressen: Für die Ausgabe der Adressen verwenden wir jetzt nicht die Formatierung `%d` sondern die Formatierung `%p`.

Zum Schluss reichern wir unser bisheriges Programm mit den entsprechenden Ausgabeanweisungen an und erhalten folgendes Programm:

[3] Ja, einfach nur `main` für die Adresse der Funktion wäre korrekter. Aber da beide Varianten `& main` und `main` funktionieren, bleiben wir aus Konsistenzgründen im Rahmen dieser Unterlagen bei ersterer.

```c
#include <stdio.h>

int main( int argc, char ** argv )
    {
        int a, b, F;
        printf( "Bitte Seite a eingeben: " );
        scanf( "%d", & a );
        printf( "Bitte Seite b eingeben: " );
        scanf( "%d", & b );
        F = a * b;
        printf( "Der Flaecheninhalt betraegt F=%d m*m\n", F );

        // Ausgaben der Speicherbelegung
        // erst die Variablen
        printf( "\nVariablen im Arbeitsspeicher\n" );
        printf( "a: Adresse=%p Groesse=%d\n", & a, sizeof(a));
        printf( "b: Adresse=%p Groesse=%d\n", & b, sizeof(b));
        printf( "F: Adresse=%p Groesse=%d\n", & F, sizeof(F));

        // jetzt die Funktionen
        printf( "\nFunktionen im Arbeitsspeicher\n" );
        printf( "main():   Adresse=%p\n", & main );
        printf( "scanf():  Adresse=%p\n", & scanf );
        printf( "printf(): Adresse=%p\n", & printf );
    }
```

Auf meinem Rechner ergibt dies folgende Ausgabe, die sich von Programmstart zu Programmstart ändern kann:

```
Variablen im Arbeitsspeicher
a: Adresse=0xbfc8fb9c Groesse=4
b: Adresse=0xbfc8fb98 Groesse=4
F: Adresse=0xbfc8fb94 Groesse=4

Funktionen im Arbeitsspeicher
main():   Adresse=0x8048474
scanf():  Adresse=0x80483a0
printf(): Adresse=0x8048390
```

Wir könnten uns auch die Adressen und Größen der (Ausgabe-) Texte ausgeben lassen: `printf( "'Bitte Seite a eingeben: ': Adresse=%p Groesse=%d\n","Bitte Seite a eingeben: ", sizeof( "Bitte Seite a eingeben: "));`

... aber das ist ziemlich lang ...

Viel Spass dabei!

Kapitel 9

Kleine Vorschau

Es liegt noch ein gutes Stück vor uns, bis wir die wesentlichen Grundlagen der abstrakten Programmierung (Implementierung) und die wesentlichen Elemente der Programmiersprache C besprochen haben. Mit der Entwicklung und Kodierung unseres ersten kleinen C-Programms haben wir aber bereits den ersten Schritt getan. Damit Ihr die nächsten Wochen nicht nur theoretisch arbeiten sondern auch direkt weiter programmieren könnt, gibt dieses Kapitel eine kurze Einführung in ein paar weitere Elemente. Diese werden wir in den beiden folgenden Skriptteilen weiter vertiefen.

9.1 Abstrakte Implementierung

Im Rahmen unseres ersten Programms haben wir die Wertzuweisung sowie die komplexe Anweisung bereits kennengelernt. Ferner haben wir über Ein- und Ausgabeanweisungen gesprochen. Hinzu kommt jetzt noch die einfache Fallunterscheidung, mittels derer man innerhalb des Programms verzweigen kann:

```
Einfache Fallunterscheidung
wenn  Bedingung erfüllt
dann  Anweisung 1
sonst Anweisung 2
```

Mit `Bedingung erfüllt` ist irgendeine logische Bedingung gemeint, die entweder zutrifft oder eben nicht. Die folgende Tabelle zeigt vier Beispiele:

Bedingung	Bedeutung
`a > 1`	Die Variable `a` hat einen Wert größer als `1`
`a + b < c`	Die Summe der Variablen `a` und `b` ist kleiner als `c`
`(a > 1) und (b > 1)`	Beide Variablen `a` und `b` sind größer als `1`
`a × a + b × b == c × c.`	$a^2 + b^2 = c^2$, ein typischer Pythagoras-Test

9.2 C-Codierung

Wiederholung: Im Rahmen unseres ersten kleinen Programms hatten wir bereits die Variablendeklaration `int a`, die Wertzuweisung an eine Variable `F = a * b` sowie die Eingabe- `scanf()` und Ausgabeanweisungen `printf()`.

Neu: Wie zu erwarten gibt es auch in der Programmiersprache C eine Entsprechung für die einfache Fallunterscheidung:

Einfache Fallunterscheidung

```
if ( Ausdruck )
{
   Anweisung ;
   ......... ;
   Anweisung ;
}
else {
       Anweisung ;
       ......... ;
       Anweisung ;
     }
```

Ja, die runden Klammern gehören dazu und *müssen* unbedingt eingetippt werden. Und eine oder mehrere obiger Anweisungen können ihrerseits wieder einfache Fallunterscheidungen sein, was man auch verschachteln nennt.

Als Bedingung steht hier der Begriff `Ausdruck`. In erster Näherung ist dies einfach eine logische Abfrage, die auch Rechenoperationen beinhalten kann. Beispiele sind: `a > b` und `a * a > b * b`. Bei diesen Abfragen sind, wie zu erwarten ist, alle vier Grundrechenarten erlaubt. Ein wenig Augenmerk erfordert die Notation der Vergleichsoperatoren. Diese sind:

Bezeichnung	Symbol	Beispiele
größer	`>`	`if ( i > 1 )` , `if ( j > i )`
kleiner	`<`	`if ( a < 123 )` , `if ( i < summe )`
größer-gleich	`>=`	`if ( i >= 0 )` , `if ( a >= i )`
kleiner-gleich	`<=`	`if ( j <= i )` , `if ( a <= -10 )`
gleich	`==`	`if ( i == 1 )` , `if ( i == 0 )`
ungleich	`!=`	`if ( i != 0 )` , `if ( i != 1 )`

Wichtiger Hinweis: Bei der Abfrage auf Gleichheit müssen unbedingt zwei Gleichheitszeichen eingetippt werden; alles andere führt zu merkwürdigen Effekten. Dies sollte eigentlich erst einmal ausreichen. Bei weiteren Fragen einfach an die Assistenten wenden.

Teil II

Zur systematischen Entwicklung von Algorithmen

Kapitel 10

Ein erster Überblick

Schauen wir noch einmal zurück zum Software Life Cycle, wie wir ihn in Kapitel 6 besprochen haben. Demnach befindet sich zwischen dem ersten Entwurf und dem eigentlichen Kodieren eine Phase, in der die Algorithmen schrittweise entwickelt werden, und zwar losgelöst von einer konkreten Programmiersprache[1]. Dabei kommt natürlich unweigerlich die Frage auf: *„Auf welche Grundzutaten kann ich denn in diesem Prozess zurückgreifen?* Genau diese Frage klären wir hier. Der weitere Aufbau dieses Skriptteils ist wie folgt:

Kapitel	Inhalt
11	Datentypen, Daten und Variablen
12	Einfache Anweisungen
13	Fallunterscheidungen
14	Schleifen
15	Illustrierende Beispiele
16	Erweiterte Flächenberechnung
17	Zusammenfassung

Die in den Kapiteln 11 bis 15 beschriebenen Elemente sind in der einen oder anderen Art und Weise in jeder Programmiersprache vorhanden. Insofern schränkt man sich bei Verwendung dieser Hilfsmittel und den Verzicht auf die Syntax einer konkreten Programmiersprache nicht all zu sehr ein. Der Lohn dafür ist, dass man eine systematische Vorgehensweise anwendet, die zu einem sehr frühen Zeitpunkt offene Fragen klärt, Entwurfsfehler sehr früh aufdeckt und damit die späteren Phasen, insbesondere das Kodieren und unnötig lange Testen, sehr stark beschleunigt. Oder einfach ausgedrückt: Es lohnt sich, die hier vorgestellten Methoden zumindest anfangs anzuwenden, denn man spart echt Zeit und vermeidet reichlich Frust, der beim ansonsten üblichen Herumprobieren entsteht. Und nach ausreichend Übung werdet ihr später wie Ingenieure und nicht wie *wanna-be's* arbeiten ...

[1] Man kann es nicht oft genug wiederholen: Das vorschnelle Eintippen einiger Programmzeilen, da man ja als Dr. Checker *sofort* „weiß", was man benötigt, ist eine Hauptursache für ineffizientes Arbeiten und schlechten Code; aber auch wenn Chefs dies meistens so wollen, wäre Planung vor Aktionismus das Richtige!

Kapitel 11

Datentypen, Daten und Variablen

Aus Kapitel 5 sei nochmals wiederholt, dass eine CPU im Prinzip nicht viel mehr kann, als Rechnen und Daten hin und her zu schieben; aber das kann sie dafür sehr schnell, einige Milliarden mal pro Sekunde. Entsprechend spielen Variablen als Datenträger eine zentrale Rolle. In der Mathematik nennt man solche Variablen meist x, y, z (skalare Variablen) oder $\vec{v}$, $\vec{x}$ (Vektoren) oder $\boldsymbol{M}$ (Matrizen). Ferner wird zu einer Variablen auch noch spezifiziert, aus welcher Zahlenmenge die erlaubten Werte sein können. In der Programmierung ist es haargenau so, nur dass man nicht von Zahlenmengen sondern von *Datentypen* spricht.

Einfache Datentypen: In den meisten Programmiersprachen gibt es die folgenden einfachen (skalare) Datentypen für Berechnungen sowie zwei weitere zur Verarbeitung von Texten jeglicher Art:

Programmierung	Mathematik
`integer`	ganze Zahlen (Z)
`unsigned integer`	natürliche Zahlen (N)
`double`	reelle Zahlen (Q/R)
`complex`	komplexe Zahlen (C) (in C nur sehr eingeschränkt)
`character`	*ein* einzelnes Zeichen
`string`	eine Zeichenkette (in C nur sehr eingeschränkt)

Und sollte man mal einen anderen Datentyp benötigen, so definiert man diesen einfach und schaut später, wie er sich in der gewählten Programmiersprache realisieren läßt.

Vektoren und Matrizen: Auch diese gibt es so gut wie in jeder Programmiersprache, nur nennt man sie hier *Felder* bzw. *Arrays.* Um sie zu verwenden, muss man zusätzlich spezifizieren, wie viele Dimensionen ein Vektor bzw. eine Matrix hat, wie viele Elemente je Dimension benötigt werden und von welchem Datentyp die einzelnen Elemente sein sollen. Die Notation sollte hier der Programmierer einfach selbst festlegen. Hauptsache, es wird klar, was gemeint ist. Mit anderen Worten: nicht religiös sondern pragmatisch vorgehen.

Strukturierte Datentypen: Aus Sicht einer Aufgaben- oder Problemstellung kann es auch sinnvoll sein, sich einen strukturierten Datentypen zu definieren. Damit meint man, dass man mehrere (einfache) Komponenten zu einem komplexen Datentypen zusammenfasst. Beispielsweise könnte eine Person einen Vornamen, einen Nachnamen, ein Gewicht und eine Körperlänge haben. Diesen Datentyp nennt man dann einfach `datentyp_person` und beschreibt die jeweiligen Komponenten. Auch hier gilt wieder, nicht religiös sondern pragmatisch vorgehen.

Adressen und Zeiger: Vielfach (insbesondere auch in der Programmiersprache C) gibt es noch einen Datentyp, den man landläufig als Adresse oder Zeiger bezeichnet und der in der Lage ist, die Adresse irgendeines Objektes (beispielsweise einer Variablen) aufzunehmen. Für die Experten bzw. Interessierten: Wenn man eine derartige Adresse mit einem anderen Datentyp zu einem strukturierten Datentyp zusammenfügt, kann man damit ganze verkettete Listen im Arbeitsspeicher verwalten. Mehr dazu aber erst in Skriptteil VII.

Dateien: Von den meisten Programmiersprachen aus kann man auch auf Dateien zugreifen. Der entsprechende Datentyp heißt meist `File`.

Kapitel 12

Einfache und komplexe Anweisungen

Auch dieser Teil der Geschichte ist schnell erzählt. Es gibt eigentlich nur folgende Anweisungen:

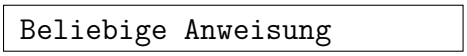
1. Variablenzuweisung (Wertzuweisung an eine Variable),
2. komplexe Anweisungen, die man im Laufe des Entwurfs bzw. der Implementierung noch weiter verfeinert,
3. Anweisungsblöcke, die aus einzelnen Anweisungen bestehen,
4. Fallunterscheidungen und
5. Wiederholungsschleifen.

Egal, was sich hinter einer konkreten Anweisung tatsächlich verbirgt, kann man sie wie folg notieren:

Schrittweise Verfeinerung

```
Beliebige Anweisung
```

Struktogramm



Im Folgenden werden die einzelnen Anweisungen näher erläutert und mittels der entsprechenden Diagramme der Methode der Schrittweisen Verfeinerung bzw. der Struktogramme illustriert. Sowohl die Anweisungen als auch deren grafische Repräsentation sind derart einfach und intuitiv, dass nur wenig erklärt werden muss.

Einfache Anweisung: In dieser Rubrik gibt es die beiden folgenden Formen:

Leeranweisung: Diese Anweisungsform besteht einfach aus nichts. Man benötigt sie manchmal, weil einige der nachfolgenden Kontrollstrukturen einfach darauf bestehen, dass dort mindestens eine Anweisung steht. Im Rahmen der Schrittweisen Verfeinerung treten Leeranweisungen nicht explizit auf, in Struktogrammen sieht man einfach einen Kasten, der leer ist ;-)

Wert-/Variablenzuweisung: Hierbei wird einer Variablen ein Wert zugewiesen:

Schrittweise Verfeinerung

```
Setze Radius = 12
```

Struktogramm

Setze Radius = 12

Komplexe Anweisung: Eine Komplexe Anweisung wird ebenso dargestellt wie eine einfache Anweisung. Nur steht dort etwas, was nicht unbedingt direkt umsetzbar ist. Dies ist immer abhängig von dem Anwendungsgebiet bzw. der gewählten Programmiersprache. Zwei Beispiele könnten sein: `koche Kaffee` oder `baue ein Auto`.

Anweisungsblock: Ein Anweisungsblock besteht aus einer Aneinanderreihung mehrerer einfacher oder komplexer Anweisungen oder weiterer Anweisungsblöcke. Diese werden einfach untereinander geschrieben. Am Beispiel einer Kreisberechnung könnte man dies wie folgt darstellen:

Schrittweise Verfeinerung

Setze U = 2 × π × R

Setze F = π × R^2

Struktogramm

Setze U = 2 × π × R
Setze F = π × R^2

Vorbemerkungen für Kontrollstrukturen: In den folgenden Beschreibungen wird immer wieder der Begriff `Anweisung` vorkommen. Hierbei kann es sich nach Belieben um eine Leeranweisung, eine einfache oder komplexe Anweisung oder gar einen Anweisungsblock handeln. Natürlich kann eine derartige Anweisung auch wiederum aus einer beliebigen Kontrollstruktur bestehen. Im folgenden Abschnitt werden hierzu verschiedene Beispiele gezeigt.

Kapitel 13

Fallunterscheidungen

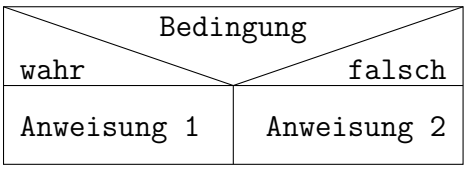
Es gibt zwei Arten von Fallunterscheidungen: die *einfache* Fallunterscheidung wählt zwischen zwei Möglichkeiten aus, die *mehrfache* Fallunterscheidung hingegen kann zwischen mehr als zwei Werten entscheiden.

Einfache Fallunterscheidung: Die einfache Fallunterscheidung überprüft eine logische `Bedingung` und je nachdem, ob sie `wahr` oder `falsch` ist, wird entweder `Anweisung 1` oder `Anweisung 2` abgearbeitet.

Schrittweise Verfeinerung

```
wenn  Bedingung
dann  Anweisung 1
sonst Anweisung 2
```

Struktogramm

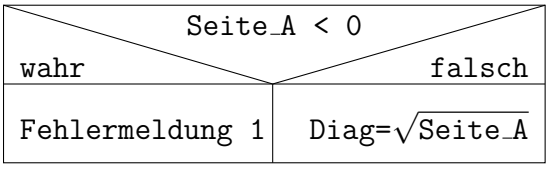


Bei der einfachen Fallunterscheidung können die drei Elemente `Bedingung`, `Anweisung 1` und `Anweisung 2` beliebige Formen annehmen. Wichtig dabei ist nur, dass die `Bedingung` entweder `wahr` (erfüllt) oder `falsch` (nicht erfüllt) ist. Wie man sieht, wird das Programm je nach Bedingung anders ausgeführt. Am Ende kommen beide Programmzweige wieder zusammen. Hierzu folgendes kleines Beispiel:

Schrittweise Verfeinerung

```
wenn  Seite_A < 0
dann  Drucke Fehlermeldung
sonst Diag = √Seite_A
```

Struktogramm



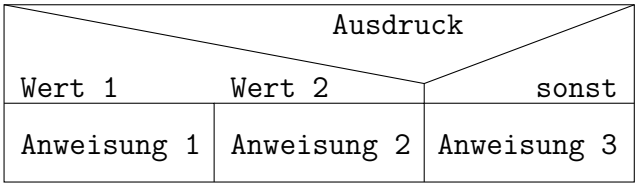
In diesem Beispiel wird im Falle einer negativen Seitenlänge eine Fehlermeldung ausgegeben bzw. die Wurzel ermittelt.

Mehrfache Fallunterscheidung: Die mehrfache Fallunterscheidung überprüft eine Variable oder einen ganzen Ausdruck bezüglich verschiedener Werte und führt bei Gleichheit die entsprechende Anweisung aus. Für den Fall, dass keiner der Werte zutrifft, wird der `sonst`-Teil ausgeführt, sofern er vorhanden ist. Die Angabe von drei verschiedenen Auswahlmöglichkeiten ist nur beispielhaft und kann nach Belieben verändert werden.

Schrittweise Verfeinerung

```
auswahl: Ausdruck
wenn Wert 1: Anweisung 1
wenn Wert 2: Anweisung 2
sonst      : Anweisung 3
```

Struktogramm

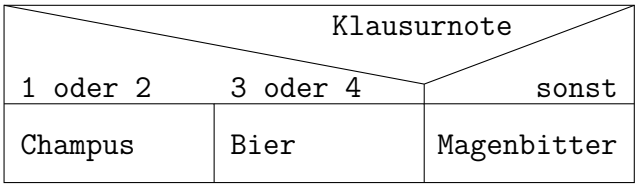


Auch dieses Programmierkonstrukt können wir mittels eines kleinen, sehr hypothetischen Beispiels etwas illustrieren:

Schrittweise Verfeinerung

```
auswahl: Klausurnote
wenn 1 oder 2: Champus
wenn 3 oder 4: Bier
sonst        : Magenbitter
```

Struktogramm



Kapitel 14

Schleifen

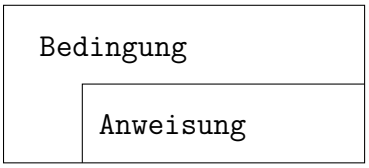
Üblicherweise werden drei Arten von Schleifen unterschieden: die `while`-Schleife, die `for`-Schleife und die `do-while`-Schleife.

While-Schleife: Bei der `while`-Schleife wird die `Anweisung`, auch *Schleifenrumpf* genannt, so lange wiederholt, wie die `Bedingung` im *Schleifenkopf* erfüllt ist. Da bei diesem Konstrukt die `Bedingung` am Anfang der Schleife steht, kann es gut sein, dass der Schleifenrumpf kein einziges Mal ausgeführt wird. Da die Bedingung *vor* dem Schleifenrumpf steht, nennt man die `while`-Schleife auch *pre-checked loop.*

`Schrittweise Verfeinerung`

```
solange Bedingung erfüllt
wiederhole Anweisung
```

`Struktogramm`

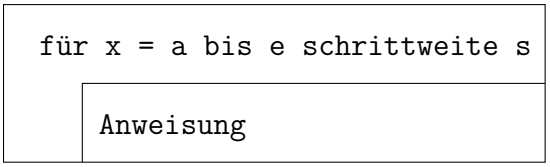


For-Schleife: Die `for`-Schleife ist so ähnlich wie die `while`-Schleife aufgebaut. Nur wird hier zunächst die Zählvariable `x` auf den Anfangswert `a` gesetzt und dann schrittweise um `s` erhöht, bis sie den Endwert `e` erreicht hat. Für jeden dieser Werte, also `x` $\in$ `[a, a+s, ..., e]` wird der Schleifenrumpf ausgeführt.

`Schrittweise Verfeinerung`

```
für x = a bis e schrittweite s
wiederhole Anweisung
```

`Struktogramm`



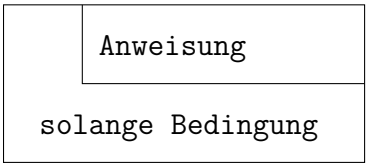
Ist bei positiver Schrittweite `s` der Anfangswert `a` bereits größer als der Endwert `e` bzw. bei negativer Schrittweite `s` der Anfangswert `a` bereits kleiner als der Endwert `e`, wird die Schleife kein einziges Mal ausgeführt (*pre-check loop*).

Do-While-Schleife: Im Gegensatz zur `while`-Schleife wird bei der `do-while`-Schleife die `Bedingung` erst am Ende überprüft, weshalb man sie auch *post-checked loop* nennt. Wichtig bei *post-checked loops* ist, dass sie in zwei Formen verwendet werden: erstens, solange *wie* die `Bedingung` erfüllt ist und zweitens *bis* die `Bedingung` erfüllt ist. Das heißt, dass im zweiten Fall (rechte Seite in den Grafiken), der Schleifenrumpf so lange ausgeführt wird, wie die `Bedingung` eben *nicht* erfüllt ist. Mit anderen Worten: Beide Fälle sind identisch, sofern man die Logik der `Bedingung` negiert formuliert.

`Schrittweise Verfeinerung`

```
wiederhole Anweisung
solange Bedingung erfüllt
```

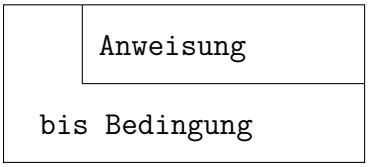
`Struktogramm`



`Schrittweise Verfeinerung`

```
wiederhole Anweisung
bis Bedingung erfüllt
```

`Struktogramm`



Kombinationen: Hier sei nochmals daran erinnert, dass in allen Kontrollstrukturen (Fallunterscheidungen sowie Schleifen) der Begriff `Anweisung` durch jede beliebige Kombination aus beliebigen Anweisungen und beliebigen Kontrollstrukturen ersetzt werden kann.

Kapitel 15

Beispiele

Dieses Kapitel präsentiert eine Reihe von Beispielen. Zum einen dienen sie dazu, die Verwendung der einzelnen Konstrukte etwas zu illustrieren. Zum anderen sollen sie zeigen, dass der Umgang mit der Methode der Schrittweisen Verfeinerung bzw. mit Struktogrammen recht einfach, intuitiv und improvisierend möglich ist.

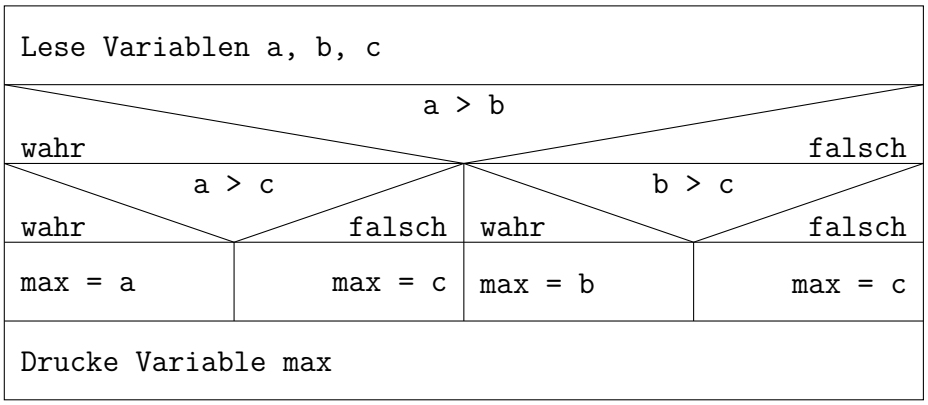
15.1 Maximum dreier Zahlen

Aufgabe: Bestimme das Maximum `max` dreier Zahlen `a`, `b` und `c`.

Eine mögliche Variante sieht als Struktogramm wie folgt aus:

Bestimme das Maximum dreier Zahlen

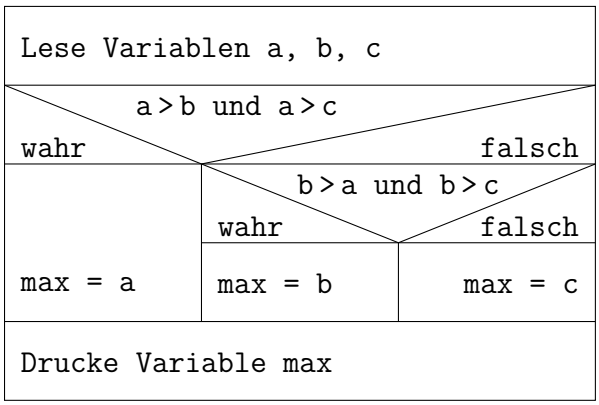
Variablen: Typ Integer: a, b, c, max



Man hätte ebenso ganz anders vorgehen können, das Resultat wäre ebenso gut. Eine dieser Möglichkeiten präsentieren wir auf der nächsten Seite.

```
Das Maximum dreier Zahlen
Variablen: Typ Integer:
           a, b, c, max
Lese Variablen a, b, c
wenn  a > b und a > c
dann  max = a
sonst wenn  b > a und b > c
      dann  max = b
      sonst max = c
Drucke Variable max
```

```
Das Maximum dreier Zahlen
Variablen: Typ Integer:
           a, b, c, max
```

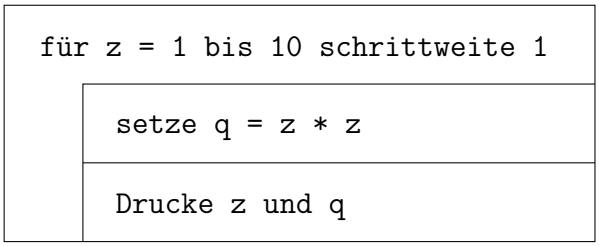


15.2 Drucke die Zahlen von eins bis zehn

Aufgabe: Drucke die Zahlen von eins bis zehn und deren Quadrate

```
Drucke 10 Zahlen und deren
Quadrate
Variablen: Typ Integer: z, q
für z = 1 bis 10 schrittweite 1
wiederhole setze q = z * z
           Drucke z, q
```

```
Drucke 10 Zahlen und deren
Quadrate
Variablen: Typ Integer: z, q
```



Bei dieser `for`-Schleife nimmt die Variable `z` nacheinander die Werte von 1 bis 10 an. Die Variable `q` wird jeweils auf das Quadrat der Zahl `z` gesetzt. In der nächsten Anweisung, also am Ende jedes Schleifendurchlaufes, werden beide Variablen ausgegeben.

15.3 Tagesplan für einen warmen Sommertag

Aufgabe: Erstelle einen Tagesplan für einen warmen Sommertag

Auch diese Aufgabe ist eigentlich wieder einfach und unkompliziert zu lösen:

```
Planung eines Sommertages
Variablen: keine
solange die Sonne scheint
wiederhole eine Stunde
           Sonne tanken
           einen Kaffee trinken
```

Planung eines Sommertages

Variablen: keine

solange die Sonne scheint	
	eine Stunde Sonne tanken
	einen Kaffee trinken

Hier haben wir eine `while`-Schleife genommen, denn es könnte ja schon am frühen Morgen regnen, und da wollen wir ja nicht an den Strand.

15.4 Shoppen mit limitiertem Konto

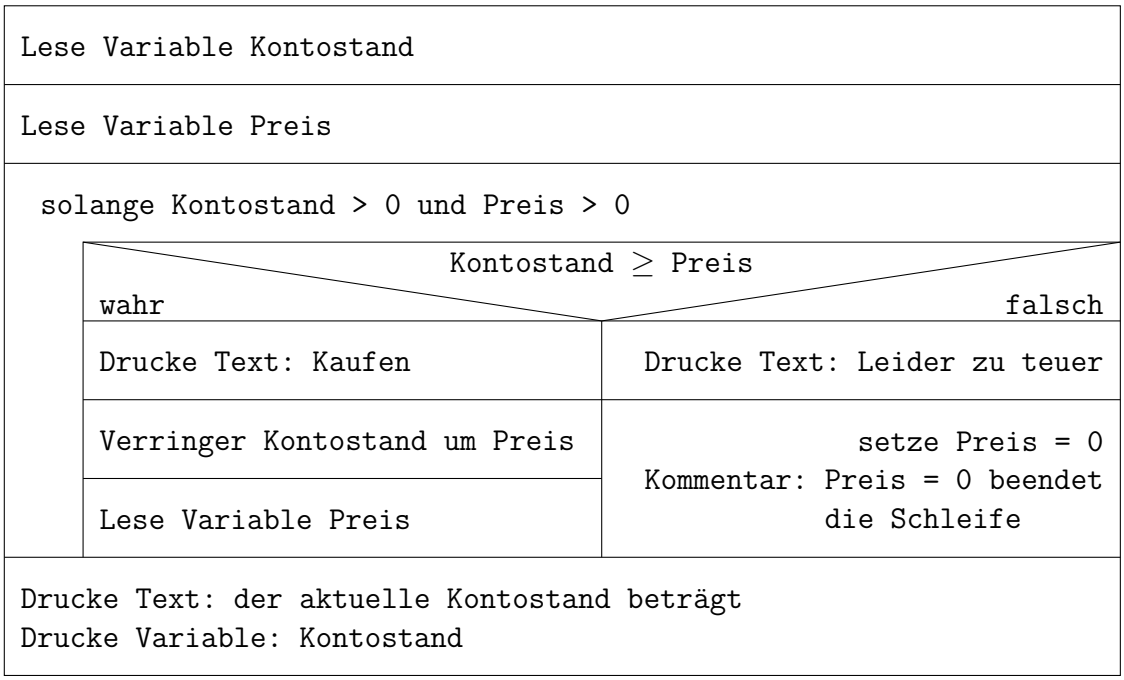
Aufgabe: Beim Shoppen ist darauf zu achten, dass der Betrag auf dem Konto bzw. der Kreditkarte limitiert ist. Der Betrag auf dem Konto darf nicht negativ werden.

Lautes Denken: Ok, wir müssen erst wissen, wieviel Geld auf dem Konto ist. Dann müssen wir für jedes Produkt den Preis einlesen und schauen, ob noch genug Geld auf dem Konto ist. Falls ja, können wir kaufen, falls nicht, haben wir Trauer. Da wir nicht wissen, wie viele Dinge wir möglicherweise kaufen wollen, benötigen wir eine der beiden normalen Schleifen (also nicht die Zählschleife). Die Schleife und damit das Programm sollen wir auch zum Ende bringen können. *Ad hoc Entscheidung:* Programmende wenn entweder das Konto leer ist oder wenn ein negativer Preis eingegeben wurde. *Eine* mögliche Implementierung eines derartigen Algorithmus ist auf der folgenden Seite in Form eines Struktogramms dargestellt.

Anmerkungen: Wichtig ist, dass am Ende der Schleife der Preis erneut (also für das nächste Produkt) eingelesen wird; sonst hätte man immer den gleichen Preis im Programm. Was würde passieren, wenn wir dies nicht täten? Auch hätten wir hier das Programm anders gestalten können. Eine gute Übung wäre jetzt, das Programm so umzustrukturieren, dass es auch in eine *post-checked loop* passt. Ferner sollten wir die Schleifenbedingung noch an die Realität anpassen, denn sonst haben wir eigentlich eine Endlosschleife. Wir könnten beispielsweise abbrechen, wenn es später als 20 Uhr ist. Viel Spass!

Shoppen gehen mit limitiertem Konto

Variablen: Typ Integer: Kontostand, Preis



Kapitel 16

Erweiterte Flächenberechnung

Bereits in Kapitel 6 haben wir ganz informell, ohne Kenntnisse von Struktogrammen und weiteren Möglichkeiten, ein Programm zur Berechnung der Fläche eines Rechtecks entwickelt. Nun wissen wir mehr und können auch mehr machen. Zunächst wiederholen wir aber nochmals das bisher entwickelte Programm:

```
Fläche eines Rechtecks
Variablen: Integer: a, b, F
Einlesen der Seite a
Einlesen der Seite b
Berechne Fläche F = a * b
Ausgabe des Flächeninhaltes F
```

```
Einlesen der Seite x
Ausgabe des Textes:
Bitte Wert für Seite x eingeben
Lese Wert für Seite x
```

```
Ausgabe des Flächeninhaltes F
Ausgabe des Textes: Der Flächeninhalt beträgt:
Ausgabe des Wertes von F
```

Wir könnten jetzt noch die Notation anpassen. Aber dazu besteht kein Grund, da es sich ja eher um eine informelle Methode handelt. Aber es lohnt sich vielleicht noch, das Programm zu erweitern. In Kapitel 6 hatten wir spezifiziert, dass für die Seiten `a` und `b` nur Werte größer als null sinnvoll sind. Ein derartiger Test ist nun mit der gelernten Fallunterscheidung echt *easy* zu realisieren und brauchen wir hier nicht extra aufzuschreiben. Die Frage ist nur, wie sich das Programm bei fehlerhaften Eingaben verhalten soll. Möglichkeit 1: es gibt eine Fehlermeldung aus und beendet sich. Möglichkeit 2: es gibt eine Fehlermeldung und liest wiederholt so lange Werte ein, bis der Nutzer einen sinnvollen Wert eingegeben hat.

Möglichkeit 1 ist einfach realisiert: man muss das Einlesen in eine entsprechende Fallunterscheidung einbetten. Sollte der Wert korrekt sein, wird weiter gemacht, andernfalls wird eine Fehlermeldung ausgegeben. Ferner muss der Platzhalter `x` durch jeweils `a` bzw. `b` ersetzt werden, wobei obiger Teil `Einlesen von Seite x` immer noch verwendet wird:

```
Fläche eines Rechtecks
Variablen: Integer: a, b, F
Einlesen der Seite a
wenn  Seite a > 0
dann  Einlesen von Seite b
      wenn  Seite b > 0
      dann  setze F = a * b
            Ausgabe des Flächeninhaltes F
      sonst Fehlermeldung: Seite b muss größer als null sein
sonst Fehlermeldung: Seite a muss größer als null sein
```

Für die zweite Möglichkeit müssen wir den Eingabeteil entsprechend ändern. Aber das ist nach dem bisher Gelernten relativ einfach. Den vorhandenen Hauptteil brauchen wir jetzt nicht mehr zu ändern, da aus dem Eingabeteil nur sinnvolle Werte kommen können:

```
Einlesen einer Seite x
Ausgabe des Begrüßungstextes: Bitte Wert für Seite x eingeben
wiederhole
   Lese Wert für Seite x
   wenn Seite x ≤ 0
   dann Ausgabe des Textes: Sorry, Seite muss größer als null sein!
        Ausgabe Text: Bitte einen korrekten Wert eingeben
bis Seite x > 0
```

Kapitel 17

Abschluss

Dieses Kapitel fasst nochmals die wesentlichen Punkte dieses zweiten Skriptteils zusammen.

War's das schon? Ja, im Wesentlichen war's das tatsächlich. Das ist alles, fast alles, viel kommt an Elementen nicht mehr hinzu.

„Dann haben wir den Rest des Semesters also frei (freu!)?“ Sorry, nicht ganz. Wir haben jetzt nur die Grundelemente besprochen, die wir im Rahmen der Programmierung so verwenden können. Das ist in etwa so, als würde ein Lehrling auf der Baustelle nun wissen, dass es prinzipiell mal Sand, Steine, Stahl, Beton, Holz, Fenster und Dachziegel gibt. Die Kunst besteht nun darin, aus diesen Grundzutaten alle möglichen Gebäude, angefangen bei einer Hundehütte, über einen Bungalow bis hin zum Wolkenkratzer, zu bauen. Dies muss man erlernen und erfordert, dass man sich den Herausforderungen stellt, sich die Übungsaufgaben vornimmt und diese auch am Rechner bearbeitet. Dann wird nachher auch 'was 'draus.

In den weiteren Kapiteln geht es entsprechend um die Umsetzung der Grundelemente in die Programmiersprache C, das Erlernen komplexer Datenstrukturen, das Erlernen von Basisalgorithmen sowie das Realisieren dynamischer Datenstrukturen.

Warum derartige Struktogramme? Wie eingangs gesagt, geht es bei den Struktogrammen nicht darum, Euch zu quälen. Vielmehr geht es um ein *Hilfsmittel*, um Eure Gedanken zu strukturieren. Ferner helfen Struktogramme dahingehend, als dass der Programmablauf nur von oben nach unten möglich ist. Dadurch hat man eine gute Möglichkeit sogenannten Spaghetti-Code zu vermeiden, der von einem großen Durcheinander geprägt ist.

Also, wenn's um die Übungsaufgaben geht, nicht gleich los tippen, sondern bei den ersten Übungspaketen erst mal ein kleines Struktogramm zeichnen (oder die Methode der Schrittweisen Verfeinerung ohne Kästchen verwenden), bis Ihr die Basiselemente halbwegs im Griff habt. Mit etwas Übung geht diese strukturierte Programmierung ins Blut über und ihr werdet es einfach im Vorbeigehen ohne Papier und Bleistift hin bekommen.

Wie wir nun gesehen haben, gibt es gar nicht so viele verschiedene Zutaten. Natürlich liegt wie immer die Tücke im Detail. Aber davon darf man sich nicht verrückt machen lassen. Einfach mal anfangen. Bei den benötigten Variablen kann man einfach mal das nehmen, was man meint zu brauchen. Bei den Anweisungen ist es ein wenig anders. Hier können wir zwischen folgenden Alternativen auswählen:

1. Leeranweisung
2. Variablenzuweisung
3. ein- oder mehrfache Fallunterscheidung
4. eine Wiederholungsschleife
5. eine komplexe Anweisung, die durch weitere *Verfeinerungen* weiter aufgedröselt wird, bis man auf die vorhandenen einfachen Anweisungen zurückgreifen kann.

Teil III

Die Programmiersprache C: ein Überblick

Kapitel 18

Ein paar Vorbemerkungen zu C

Das Ziel dieses Vorlesungs- bzw. Skriptteils ist es, möglichst rasch einen ersten Überblick über die Programmiersprache C zu bekommen. Um dabei nicht zu viel Zeit zu verlieren, werden wir an einigen Stellen einige Details weggelassen bzw. die Sachverhalte stark vereinfachen; natürlich werden wir die Details zu einem späteren Zeitpunkt besprechen, fest versprochen. Der tiefere Grund hierfür liegt darin, dass Ihr möglichst schnell mit der praktischen Programmierarbeit anfangen könnt. Nur so bleibt Euch genügend Zeit intensiv zu üben.

Im Gegensatz dazu behandeln einige andere Lehrveranstaltungen und viele Leerbücher erst jeden Punkt *en Détail,* bevor sie praktisch ans Werk gehen. Auch diese Herangehensweise hat natürlich seine Berechtigung, erscheint uns aber für unsere Zuhörerschaft nicht der optimale Weg zu sein. Wir versuchen, hier einen möglichst ballastfreien Überblick zu geben. Aber auch das ist recht mühsam und energieaufzehrend. Also, habt etwas Ausdauer und Durchstehvermögen.

Vorab: Die Elemente einer Programmiersprache lassen sich grob gesagt in zwei Kategorien unterteilen. Dies sind zum einen die Daten einschließlich der dazugehörigen Variablen und Konstanten und zum anderen Anweisungen, wozu vor allem die Kontrollstrukturen wie Schleifen und Fallunterscheidungen zählen.

Bereits in Skriptteil II haben wir einen ersten Überblick über die abstrakte Programmierung gegeben. Die gute Nachricht ist, dass diese Elemente wieder sehr ähnlich von der Programmiersprache C angeboten werden. Die schlechte Nachricht ist, dass die Tücke wie so oft im Detail liegt. Es kommt auf fast jeden Tastendruck an. Hier bleibt nichts anderes übrig, als sich alles genau anzusehen und sehr diszipliniert zu programmieren. Die Tabelle auf der folgenden Seite gibt einen Überblick über die Themen, die wir in den einzelnen Kapiteln dieses Skriptteils behandeln werden.

In diesem Skriptteil beschränken wir uns auf die Grundlagen, die in der Programmiersprache aus sehr vielen Details bestehen. Trotz der Detailfülle haben wir versucht, die einzelnen

Kapitel	Inhalt
19	Kommentare und richtiges Formatieren
20	Syntaxdiagramme und (Variablen-) Namen
21	Datentyp `int`
22	Ausdrücke und Formeln
23	Generelles zu Anweisungen
24	Einfache Fallunterscheidung
25	Mehrfache Fallunterscheidung
26	Die `while`-Schleife
27	Die `for`-Schleife
28	Die `do-while`-Schleife
29	Die ASCII-Tabelle zur Definition von Zeichen
30	Datentyp `char`
31	Klassifikation von Zeichen
32	Datentyp `double` für „reelle“ Zahlen
33	Arrays: Eine erste Einführung
34	Qualitätskriterien

Kapitel recht kompakt zu halten. Dadurch erfüllt jedes Kapitel zwei Aufgaben: Erstens bietet es einen ersten Überblick über den Stoff und zweitens dient es später als Nachschlagewerk für spezielle Inhalte. Aufgrund dieser Ausrichtung sollte kein Programmieranfänger von sich erwarten, dass er alles beim ersten Lesen versteht. Die weitaus komplexeren Inhalte wie Arrays (Felder), Zeichenketten, Funktionen und dynamische Datenstrukturen behandeln wir erst in den Skriptteilen V und VII.

„Wie, das soll alles sein? Was ist mit solchen Anweisungen wie ***`main`****,* ***`printf`*** *und* ***`scanf`****? Diese Anweisungen hatten wir schon und sie müssen doch irgendwo auftauchen!“* Ja, nee, wäre jetzt die Antwort. Wir haben sie zwar bisher wie Anweisungen verwendet, es sind aber gar keine. Es sind nämlich Funktionen, wie wir sie aus der Mathematik kennen: $f(x) = \sin(x)$ oder $f(x) = a_2x^2 + a_1x + a_0$. Das Schreiben von Funktionen und deren Verwendung ist eigentlich recht einfach. Doch muss man bereits einiges über C wissen, um es richtig zu machen. Daher besprechen wir Funktionen erst in den Kapiteln 44 und 47.

Im Gegensatz zu vielen anderen Programmiersprachen kann man in der Programmiersprache C Funktionen aufrufen und ihr Ergebnis (Rückgabewert) sofort wieder vergessen. Und wer es nicht glaubt, der lasse sich einfach mal den Rückgabewert eines Aufrufs von `printf()` ausgeben: Dazu den Rückgabewert einer Variablen zuweisen und wie gewohnt ausgeben. Der Compiler jedenfalls denkt sich: *If they don't care, why should I . . .?*

Abschließen werden wir diesen Skriptteil mit einigen grundlegenden Gedanken zum Thema Qualität von Software, aus denen sich ein paar Entwicklungsrichtlinien ergeben.

Kapitel 19

Lesbarkeit: einiges zu Leerzeichen, Leerzeilen und Kommentaren

Wie bereits im ersten C-Programm (Kapitel 7) gesehen, ist ein Quelltext – auch wenn alles noch recht unverständlich – doch eine recht lesbare Angelegenheit. Zumindest stehen in so einem Progamm die üblichen Zeichen und Wörter, die wir auch in unseren Büchern haben. Zur Verbesserung der Lesbarkeit eines Programms (eigentlich Quelltextes) können in so einem Programm beliebig viele Kommentare, Leerzeichen und Leerzeilen eingefügt werden. Und je besser ein Programm strukturiert und kommentiert ist, um so leichter lassen sich Fehler finden und die eigenen Ideen später wieder verstehen.

19.1 Kommentare

In der Programmiersprache C gibt es zwei Sorten von Kommentaren, welche mit einem doppelten Schrägstrich `// Kommentar` und welche der Form `/* Kommentar */`. Alle Kommentare werden vom C-Präprozessor, der ersten Stufe des Compilers (siehe auch Kapitel 39), entfernt, sodass sie für die weiteren Stufen des Compilers nicht mehr sichtbar sind. Mit anderen Worten: die Kommentare sind für uns als Programmierer und nicht für den Compiler gedacht.

Kommentare der Form `// Kommentar`: Bei der ersten Form fängt ein Kommentar an, wenn zwei Schrägstriche (Divisionszeichen) direkt aufeinander folgen, d.h., wenn zwischen ihnen nichts anderes steht. Dieser Kommentar geht bis zum Zeilenende und hört definitiv genau dort auf. Will man mehr als zwei Zeilen Kommentar schreiben, muss man diesen in jeder Zeile erneut mittels `//` beginnen, oder die Zeile mit einem Backslash \ beenden. Die folgenden Beispiele zeigen den Quelltext und das, was der Präprozessor daraus macht:

C-Quelltext

```
1 //#include <stdio.h> Keine Ein/Ausgabe
2
3 int main()           // Hauptprogramm
4     {                // Anfang main()
5         int a, b, F; // Variablen
6         F = a / b;   // Berechnung
7         F = a // b;  // Berechnung
8         F = a / / b; // Berechnung
9     }                // Hauptprogramm
```

Nach dem Präprozessor

```
1
2
3 int main()
4     {
5         int a, b, F;
6         F = a / b;
7         F = a
8         F = a / / b;
9     }
```

Kommentare der Form `/* Kommentar */`: Diese Kommentare sind komplett anders. Sie beginnen mit der Zeichensequenz `/*` und müssen durch die Sequenz `*/` explizit beendet werden, egal wie viele Zeilen später dies passiert. Hier wieder ein paar Beispiele:

C-Quelltext

```
1 /*#include <stdio.h> Kein I/O */
2
3 int main()
4     {   /* Anfang Main */
5         int a, /* b, kein b */ F;
6         /* keine Berechnung
7         F = a * b;            */
8         F = a * b;  /* aber hier */
9     }   /* Ende Main */
```

Nach dem Präprozessor

```
1
2
3 int main()
4     {
5         int a, F;
6
7
8         F = a * b;
9     }
```

Geschachtelte Kommentare: Kommentare können auch ineinander verschachtelt werden. Aber, das hat keinen Effekt! Kommentare, die mit `//` anfangen, gehen *genau* bis zum Zeilenende, egal, was da noch kommt. Und Kommentare, die mit `/*` anfangen, gehen bis zum nächsten `*/`, egal, was zwischen diesen beiden Enden steht. Aber siehe selbst:

C-Quelltext

```
1  // /* */ geschachtelt mit //
2     la-10 /* la-11 la-12
3     la-20 // */ la-21 la-22
4
5  // // geschachtelt mit /* */
6     la-30 // la-31 /* la-32
7     la-40 la-41 */ la-42
8
9  // /* */ geschachtelt mit /* */
10    la-50 /* la-51 /* la-52
11    la-60 */ la-61 */ la-62
```

Nach dem Präprozessor

```
1
2     la-10
3                 la-21 la-22
4
5
6     la-30
7     la-40 la-41 */ la-42
8
9
10    la-50
11              la-61 */ la-62
```

Wie so oft gilt folgendes: Nehmt diejenigen Kommentare in derjenigen Form die euch gefällt. Aber bleibt in eigenem Interesse konsistent, da dies die Fehlersuche drastisch vereinfacht.

19.2 Leerzeichen und Leerzeilen

Im Rahmen der C-Programmierung sind Leerzeichen und Leerzeilen nahezu komplett[1] überflüssig; sie sind nur für uns da, weil sie den Quelltext eines Programms unwahrscheinlich lesbar machen können. Dies betrifft vor allem das Einrücken von Zeilen: Es erhöht nicht nur die Lesbarkeit sondern hilft auch Euch und den Assistenten bei der Fehlersuche. Hier ein paar Beispiele:

So 'isses fein

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5         int i;
 6         i = 1;
 7         if ( i < 0 )
 8            printf( "negativ\n" );
 9         else printf( "positiv\n" );
10     }
```

Schon nicht mehr schön

```
1 #include <stdio.h>
2
3 int main(int argc,char **argv)
4 {
5 int i;
6 i = 1;
7 if ( i < 0 )
8 printf( "negativ\n" );
9 else printf( "positiv\n" );
10 }
```

Gruselig

```
1 #include <stdio.h>
2 int main(int argc,
3 char**argv)
4 {
5 int i;
6 i=1;
7 if(i<0)
8 printf("negativ\n");
9 else printf("positiv\n");
10 }
```

Ganz schrecklich

```
1 #include <stdio.h>
2 int main(int argc,char
3 **argv){int i;i=1;if(i<0)
4 printf("negativ\n"); else
5 printf("positiv\n");}
```

The worst

```
1 #include <stdio.h>
2 int main(int argc,char**ar\
3 gv){int i;i=1;if(i<0) prin\
4 tf("negativ\n");else print\
5 f("positiv\n");}
```

Und für diejenigen, die es einfach nicht glauben wollen: Alle fünf Varianten lassen sich vom Compiler fehlerfrei übersetzen und führen zum gleichen Resultat. Und wer eine Beratung benötigt, der komme bitte mit der ersten Variante.

Einrücknormen: Davon gibt es viele. *„Und welche sollen wir nehmen?“* Ist uns egal. Nehmt diejenige, mit der Ihr am besten zurecht kommt. Aber bleibt konsistent!

[1] Die einzige Ausnahme von dieser Regel sind die Textausgaben `"..."`; die hier eingefügten Leerzeichen sollen in der Regel auch auf dem Bildschirm ausgegeben werden.

Kapitel 20

Syntaxdiagramme: Beispiel Namen

Für jede (natürliche) Sprache gibt es Rechtschreib- und Grammatikregeln. Diese legen fest, ob ein Satz richtig oder falsch ist, wie jeder Schüler leidlich erfahren hat. Bei Programmiersprachen ist dies ebenso. Die Regeln werden anhand von Syntaxregeln sowie einiger Kontextbedingungen festgelegt. Aus verschiedenen Gründen geschieht dies bei Programmiersprachen sehr formal. Einer dieser Gründe ist: Im Gegensatz zu umgangssprachlichen Formulierungen können formale Beschreibungen eindeutig formuliert und darüberhinaus von anderen Programmen verarbeitet werden, die ihrerseits daraus beispielsweise einen Compiler automatisch erzeugen können. Eine der eher formalen Methoden nennt sich Syntaxdiagramme, die in diesem Kapitel recht informell eingeführt werden.

20.1 Zwei Begriffe

Für die formale Beschreibung einer Syntax benötigt man die beiden folgenden Begriffe:

Terminalsymbol: Das sind die Symbole (oder einfach Zeichen), die direkt hingeschrieben und nicht weiter zerlegt werden.

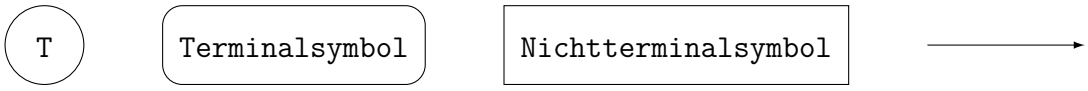
Nichtterminalsymbol: Das sind diejenigen Symbole, die man nicht hinschreibt sondern mittels weiterer Syntaxdiagramme beschreibt.

Bildlich gesprochen kann man die Terminalsymbole mit den Einzelteilen, beispielsweise mit Schrauben, Muttern und Zahnrädern, und die Nichtterminalsymbole mit den daraus gefertigten Baugruppen, beispielsweise einem Getriebeblock, vergleichen.

Wie bei einem Bauplan auch werden die einzelnen (komplexen) Sprachelemente mittels verschiedener Syntaxdiagramme solange in ihre Bestandteile zerlegt, bis sie vollständig durch die zulässigen Terminalsymbole, also ihre Einzelteile, beschrieben sind.

20.2 Grafische Elemente

Syntaxdiagramme bestehen aus drei Komponenten. Erstens: Terminalsymbole werden als Kreise oder Rechtecken mit abgerundeten Ecken dargestellt. Zweitens: Nichtterminalsymbole werden als Rechtecke mit nicht abgerundeten Ecken dargestellt. Drittens: Terminal- und Nichtterminalsymbole werden mit gerichteten Pfeilen, also Strichen mit Anfangs- und Endpunkt, miteinander verbunden. Diese Pfeile gehen von einem Rechteck zum nächsten oder beginnen bzw. enden an einem anderen Pfeil. Das war's auch schon.

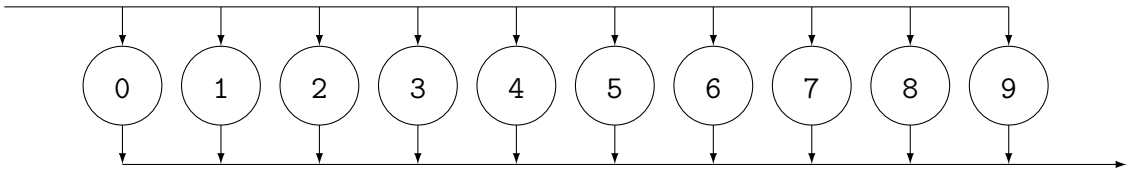


Das Lesen eines Syntaxdiagramms ist ganz einfach: Man startet beim Namen des Syntaxdiagramms (bei uns immer oben links) und geht den Pfeilen entlang, die man zuvor ausgewählt hat. Eine vorhandene Zeichenfolge ist dann korrekt, wenn es für diese einen Weg durch das Syntaxdiagramm gibt, andernfalls ist die Zeichenfolge inkorrekt.

20.3 Beispiel: Variablennamen

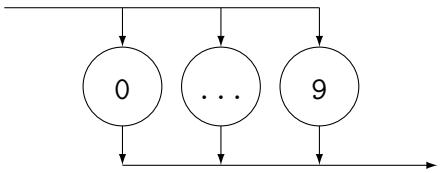
Zuerst würde man definieren, was genau eine Ziffer ist:

`Ziffer (ausführlich)`



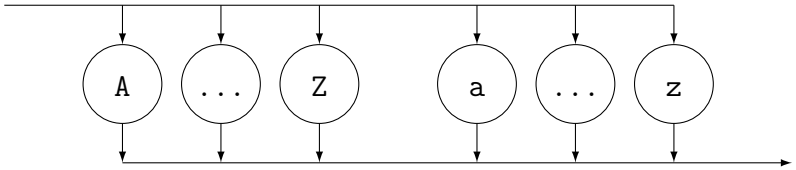
Eigentlich ziemlich trivial. Eine Ziffer ist also entweder eine `0`, eine `1`, eine `2` usw. Etwas formaler würde man dies wie folgt ausdrücken: Für das Nichtterminalsymbol `Ziffer` gibt es einen Weg (Pfade) durch eines der Terminalsymbole `0`, `1`, ..., `9`. Allerdings wirkt diese Art der formalen Beschreibung einer so trivialen Sache doch etwas übertrieben. Daher gibt es auch die folgende Kurzschreibweise, die für uns Menschen intuitiv und klar sein sollte:

`Ziffer (Kurzform)`



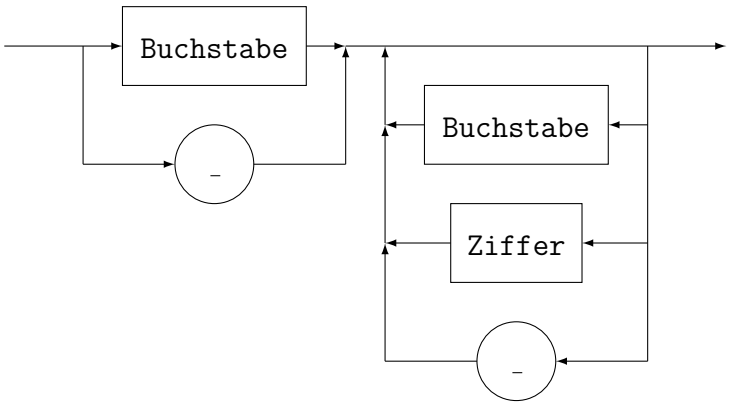
Als nächstes würde man jetzt definieren, was alles zu den erlaubten Buchstaben gehört. Da die Programmiersprache C in den USA entwickelt wurde, sind dies nur die 26 Klein- sowie Großbuchstaben, was wir in der soeben besprochenen Kurzschreibweise wie folgt darstellen können:

Buchstabe



Aufbauend auf diese beiden Syntaxdiagramme läßt sich ein in der Programmiersprache C gültiger Variablenname, im *engl.* Identifier, wie folgt beschreiben:

Identifier



Gemäß dieses Syntaxdiagramms besteht ein Variablenname (Identifier) aus einem Buchstaben oder einem Unterstrich und wird anschließend optional von weiteren Buchstaben, Ziffern oder Unterstrichen gefolgt.

Korrekte Beispiele: `Peter`, `heute`, `hEUTe`, `_hallo`, `ich_bin_bloed`, `_`, `a0` und `a0_12`.

Alle Beispiele fangen entweder mit einem Buchstaben oder Unterstrich an. Also kommt man immer an das Ende der ersten Alternative. Anschließend folgen nur Unterstriche, Buchstaben oder Ziffern, was sich immer durch den rechten Teil des Syntaxdiagramms durch wiederholtes „Abfahren“ der nach vorne gerichteten Schleife ableiten läßt.

Fehlerhafte Beispiele:

Für keinen der folgenden, fehlerhaften Namen gibt es einen Weg durch das Syntaxdiagramm:

```
1 0_ahnung          // darf nicht mit einer Ziffer anfangen
2 peter*ingo        // stern ist nirgends erlaubt
3 hallo+            // auch ein pluszeichen nicht
4 meeting@home      // das at-zeichen (@) auch nicht
```

20.4 Kontextregeln

In den Kontextregeln werden weitere Zusatzinformationen gegeben. In unserem Beispiel könnte dies sein, ob zwischen Groß- und Kleinbuchstaben unterschieden wird. In der Programmiersprache C wird das tatsächlich auch so gemacht, d.h. `Maria`, `maria`, `MARIA` und `MaRiA` sind vier verschiedene Identifier.

Kapitel 21

Datentyp `int` für ganze Zahlen

`int` ist einer *der* grundlegenden Datentypen fast aller Programmiersprachen. Er erlaubt das Abspeichern und Rechnen mit ganzen Zahlen. Beim Rechnen werden üblicherweise auch Konstanten benötigt. Diese kann man in drei verschiedenen Formaten angeben: Oktal (Basis acht), Dezimal (Basis zehn) und Hexadezimal (Basis 16).

21.1 Verwendung

C-Syntax

```
int a, b, c;
a = 1;
c = a * b;
```

Abstrakte Programmierung

```
Variablen: Typ Integer: a, b, c
setze a = 1
setze c = a * b
```

Hinweise: Bei Verwendung von `int`-Variablen und `int`-Zahlen (Konstanten) sollte folgendes beachtet werden:

1. Bevor eine Variable benutze wird, muss sie definiert[1] werden. Dies passiert in obigem Beispiel in der ersten Zeile.
2. Eine definierte Variable kann immer nur innerhalb der geschweiften Klammern verwendet werden, in denen sie auch definiert wurde.

21.2 `int`-Konstanten und interne Repräsentation

Wie eingangs erwähnt, können `int`-Zahlen (Konstanten) durch den Programmierer in drei verschiedenen Formaten angegeben werden. Aber egal welches Format man auch immer wählt, im Arbeitsspeicher kommt es immer zur gleichen Bit-Repräsentation. Mit anderen

[1] Es geht auch noch ein klein wenig anders, aber das besprechen wir erst in Kapitel 55

Worten: Die Designer der Programmiersprache C bieten uns drei verschiedene Formate an, damit wir es je nach Problemstellung möglichst einfach haben.

Aber was heißt hier Format? Damit ist immer die zugrundeliegende Basis gemeint. Wir als Menschen rechnen normalerweise im Dezimalsystem. Von hinten aus betrachtet, rechnen wir mit Einern, Zehnern, Hundertern, Tausendern usw. Im Arbeitsspeicher gibt es nur Nullen und Einsen. Entsprechend haben wir Einer, Zweier, Vierer, Achter usw.

„Und wie erkenne ich nun, welches Format eine Zahl hat?“ Ja, das ist recht einfach. Dezimalzahlen werden so eingegeben, wie wir es gewohnt sind. Die Oktalzahlen (Basis acht) erkennt man daran, dass sie immer mit einer Null anfangen. Und die Hexadezimalzahlen (Basis 16) fangen immer mit `0x` oder `0X` an. Beispiel:

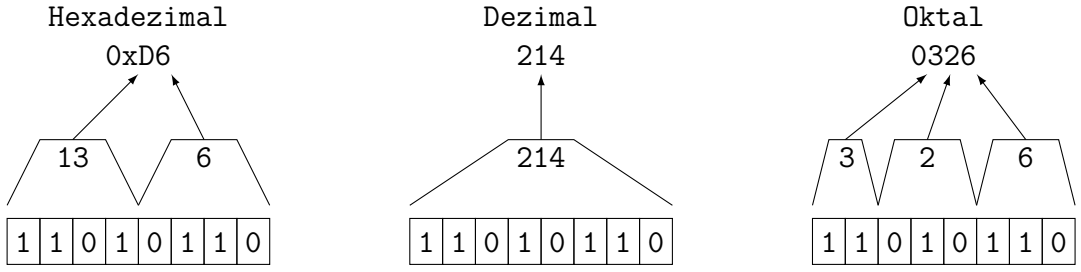
```
Hexadezimal  Dezimal  Oktal          In allen Fällen
       0x12       18    022             Wert = 18
```

„Bei Hexadezimalzahlen brauche ich doch aber 16 verschiedene Ziffer. Aber ich kenne nur zehn, nämlich die Ziffern von 0..9. Und wo kommen die anderen sechs her?“ Wieder einmal eine sehr gute Frage. Die Antwort ist ganz einfach: Die Buchstaben von `a` bis `f` haben die Werte von zehn bis 15. Dabei ist es egal, ob es sich um Klein- oder Großbuchstaben handelt.

Darstellung	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadezimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
											a	b	c	d	e	f
Dezimal	0	1	2	3	4	5	6	7	8	9						
Oktal	0	1	2	3	4	5	6	7								

So, genug der Vorrede. Folgende Grafik fasst das einfach mal alles zusammen:

Interne Repräsentation im Arbeitsspeicher



Wir hätten also `i = 0xD6;`, `i = 214;` und `i = 0326;` schreiben können und es wäre intern immer die selbe Zahl gewesen. Natürlich werden in heutigen Rechnern nicht nur acht Bits für eine ganze Zahl verwendet, sondern normalerweise 32 oder gar 64 Bits; die Beschränkung auf acht Bits war nur für die Illustration.

„Und woher weiss ich nun, wie viele Bits meine Zahlen intern haben?“ Ganz einfach, hierfür

gibt es doch die Funktion `sizeof()` (siehe auch Kapitel 37.3). Dieser Funktion kann man entweder einen Typ, eine Variable, eine Konstante oder einen ganzen Ausdruck übergeben:

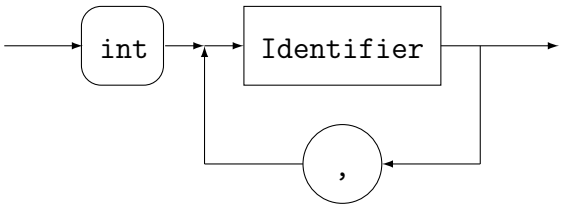
```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5         int i;
 6         printf( "Zahl der Bytes: %d\n", sizeof( int ) );
 7         printf( "Zahl der Bytes: %d\n", sizeof( i ) );
 8         printf( "Zahl der Bytes: %d\n", sizeof( 0xD6 ) );
 9         printf( "Zahl der Bytes: %d\n", sizeof( i - 1 ) );
10     }
```

Und bei mir erscheint vier mal die Zahl 4 auf meinem Bildschirm.

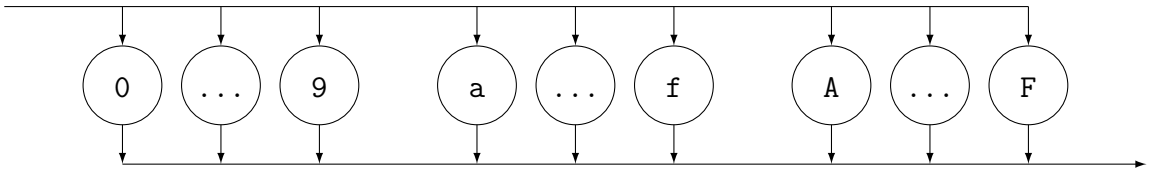
21.3 Syntaxdiagramme (vereinfacht)

Das noch etwas vereinfachte Syntaxdiagramm sieht wie folgt aus:

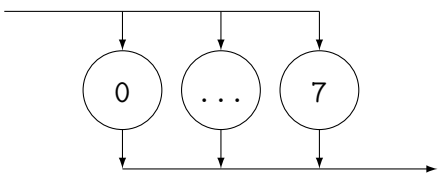
int-Definition (vereinfacht)



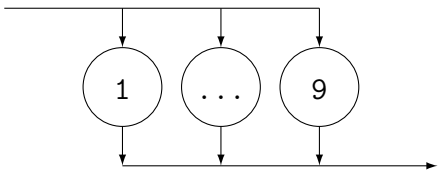
Hex-Ziffer



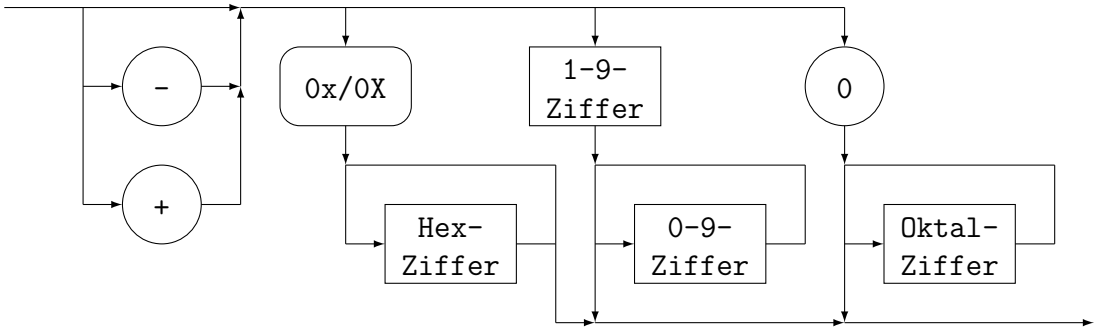
Oktal-Ziffer



1-9-Ziffer



int-Konstante



21.4 Korrekte Beispiele

```
1 // Definitionen
2
3 int i;
4 int i, j, k;
5 int summe, flag_1;
```

```
1 // Dezimal
2
3 i = 1234;
4 j = -124
5 k = j - 10;
```

```
1 // hex/octal
2
3 j = -0xABCD;
4 i = j - 012;
5 k = +0123;
```

21.5 Fehlerhafte Beispiele

```
1 integer i; // tippfehler
2 int i j;   // komma fehlt
3 int -flag; // name falsch
```

```
1 i = 0abc; // octal: 0..7
2 i = 1 2;  // 2 Konstanten
3 i = ++2;  // + zu viel
```

21.6 Ausgabe von `int`

Wie wir schon des Öfteren gesehen haben, geschieht die Ausgabe mittels der Funktion `printf()`. Die entsprechende Konvertierung lautet `%d`. Beispiel: Mittels `printf( "Zahlen: %d %d\n", i, j )` werden die beiden `%d` durch die aktuellen Werte der beiden Variablen `i` und `j` ersetzt.

Die Formatierung `%d` nimmt immer so viele Stellen in Anspruch, wie der entsprechende Wert benötigt. Zum tabellarischen Formatieren kann man auch die gewünschte Breite angeben. Dazu schreibt man diese direkt vor das `d`. Beispielsweise verwendet `printf( "i: %10d\n", i )` genau 10 Zeichen (einschließlich einem eventuellen negativen Vorzeichen),

sofern die Zahl nicht mehr Stellen benötigt. Sollte die Zahl länger als 10 Ziffern sein, wird die Ausgabe entsprechend verbreitert.

21.7 Einlesen von `int`-Werten

Wie wir bereits einige Male gesehen haben, kann man Zahlenwerte vom Typ `int` mittels der Funktion `scanf()` einlesen. Hier kann man als Formatierung `%d` angeben (weitere Möglichkeiten kann man am besten aus der Dokumentation entnehmen). Beispiel: Mittels `scanf( "%d", & i )` liest man einen Wert für die `int`-Variable `i` ein.

21.8 Definition einschließlich Initialisierung

Häufig ist es so, dass man direkt nach der Definition einige Variablen mit bestimmten Werten initialisiert. In der Programmiersprache C kann man dies zusammenfügen: Direkt hinter den Namen kann man einen Wert zuweisen. Der Compiler trennt diese beiden Dinge wieder automatisch. Hier ein paar Beispiele:

```
1 // mit initialisierung
2
3 int i = 1, j = 2;
4 int sum = i - 2;
5 int x = i * sum - j;
```

```
1 // getrennte initialisierung
2
3 int i, j, sum, x;
4 i = 1; j = 2;
5 sum = i - 2;
6 x = i * sum - j;
```

Beide Programme sind äquivalent und die gewählte Variante ist reine Geschmacksache.

21.9 Rechenoperationen und Rundungsfehler

Mit `int`-Zahlen und Variablen kann man auch rechnen. Zugelassen sind die „normalen“ Rechenarten `+`, `-`, `*` und `/`. Das Ergebnis dieser vier Rechenarten ist immer wieder eine ganze Zahl. Für die ersten drei Rechenarten ist dies unproblematisch, denn das Ergebnis kann nichts anderes als wieder eine ganze Zahl sein. Bei der Division ist dies anders: die Nachkommastellen werden *grundsätzlich, ohne jegliche Ausnahme* abgeschnitten. Beispiel: `999/1000` ergibt immer `0` und `67/17` ergibt immer `3`.

Hinzu kommt noch der Modulo-Operator-`%`, der den Divisionsrest ergibt. Beispiel `14 % 5` ergibt `4`. Mehr dazu gibt es im folgenden Kapitel.

Kapitel 22

Ausdrücke, Formeln und dergleichen

Mein Englischlehrer bekam immer so einen komischen Gesichtsausdruck, wenn er meine Klassenarbeit zurückgegeben hat. Nun ja, war meistens eine fünf. Aber das ist mit dem Begriff „Ausdruck“ nicht gemeint. Informell ausgedrückt handelt es sich um Formeln, wie man sie aus der Mathematik kennt.

22.1 Informelle Beschreibung

Wenn wir an Ausdrücke denken, sind sie in erster Näherung das, was wir bei Formeln auf der rechten Seite einer Gleichung haben. Links vom Gleichheitszeichen steht dann ein Funktionsname oder eine Zielvariable. Im Sinne der Programmiersprache C gehören die folgenden Bestandteile zu Ausdrücken (Formeln):

Bezeichnung	Beispiele
Konstanten	`123, -14`
Variablen	`i, j, summe, rest`
Funktionen	`sin(x), sqrt(x)`
Arithmetische Ausdrücke	`1 + 3*i, (i - sin(x))*(j - rest)`
Vergleiche	`a > b, i < -1`
Logische Ausdrücke	`(a == 1) && (b == 2)`
Zuweisung	`x = 1`

Diese einzelnen Formen werden wir im folgenden etwas genauer erklären.

Konstanten: Hierunter ist genau das zu verstehen, was man sich denkt. Umgangssprachlich: alle Zahlen, später dann auch einzelne Zeichen wie z.B. `'A'` und ganze Zeichenketten wie z.B. `"das ist ja mega cool"`. Darüber hinaus ist es so, dass der Compiler auch schon das ausrechnet, was er selbst machen kann. Beispielsweise ersetzt er den Ausdruck

`2 + 3*4` von sich aus durch die Konstante `14`, da sich auch bei mehrmaligem Nachrechnen das Ergebnis nicht ändern wird.

Variablen: Jede Variable für sich alleine ist bereits ein Ausdruck. Schreibt man den Variablennamen hin, liest die CPU die entspechende Speicherstelle aus und nimmt den dortigen Wert für die weiteren Berechnungen.

Funktionen: Funktionen wie beispielsweise `sin(x)` und `log(x)` sind rein intuitiv Formeln oder Teile davon. Und insofern gelten sie programmiertechnisch auch als Ausdrücke. Und wie weiter oben schon gesagt, trifft das auf die eingangs als Anweisungen bezeichneten Funktionen wie `printf()` und `scanf()` zu. Richtig gelesen, bei `printf()`, `scanf()` etc. handelt es sich um Funktionen. Ja sogar `main()` ist eine Funktion.

Arithmetische Ausdrücke: Diese Ausdrücke kennen wir aus unserer Schulmathematik. In der Programmiersprache C gibt es die vier üblichen Grundrechenarten `+`, `-`, `*` und `/` sowie den Modulo-Operator `%`. Auf beiden Seiten dieser Operatoren können wieder vollständige Ausdrücke, also Konstanten, Variablen, Funktionen usw. stehen. Einzelne Beispiele hatten wir bereits in unseren diversen Programmbeispielen sowie in obiger Tabelle.

Vergleiche: Der Vergleich von Variablen mit anderen Variablen oder konstanten Werten eröffnet ganz neue Möglichkeiten und macht die Sache recht spannend. Hier bietet die Programmiersprache C die folgenden sechs Möglichkeiten:

Bezeichnung	Symbol	Beispiele
größer	`>`	`i > 1` , `j > i`
kleiner	`<`	`a < 123` , `i < summe`
größer-gleich	`>=`	`i >= 0` , `a >= i`
kleiner-gleich	`<=`	`j <= i` , `a <= -10`
gleich	`==`	`i == 1` , `i == 0`
ungleich	`!=`	`i != 0` , `i != 1`

Wichtig dabei ist, dass die vier Symbole `>=`, `<=`, `==` und `!=` ohne Leerzeichen geschrieben werden. Also, `>=` und nicht `> =`.

Logische Ausdrücke: Es sollte jedem klar sein, dass die eben beschriebenen Vergleichsoperatoren richtig oder falsch sind. D.h., sie sind ihrerseits bereits logische Ausdrücke. Diese logischen Ausdrücke kann man mittels der folgenden logischen Operatoren miteinander verknüpfen:

Bezeichnung	Symbol	Beispiel
und	`&&`	`(i > 1) && (j == 1)`
oder	`||`	`(i > 5) || (i < -1)`
nicht	`!`	`! (i == 0)`

Zuweisung: Ja, sogar eine Zuweisung wie beispielsweise `x = 1` ist ein Ausdruck! Als Programmieranfänger weiß man in der Regel nicht, wie man das verstehen soll. Ist auch subtil. Dadurch kann man auch schreiben `i = (j = 1);` oder sogar `i = j = 1;` und beide Variablen `i` und `j` werden auf den Wert `1` gesetzt. Aus diesem Sachverhalt ergeben sich noch weitere Konsequenzen, auf die wir aber erst im fortgeschrittenen Teil ab Kapitel 43 eingehen werden.

Klammern `()`: Gerade am Anfang weiß man nicht, was, wann zu welchem Zeitpunkt ausgerechnet wird. Insofern ist es immer mal hilfreich, runde Klammern `()` zu setzen, wenn in einem Ausdruck die Operatoren gemischt werden. In obigen Beispielen haben wir das auch so gemacht, auch wenn die Klammern nicht unbedingt notwendig sind.

Liste von Ausdrücken und der Komma-Operator: Wenn wir einzelne Ausdrücke wie beispielsweise `i = 1, j = 2, k = 3` mittels Kommas aneinanderreihen dann ist dies wieder ein Ausdruck. Dies wird zunächst bei der `for`-Schleife in Kapitel 27 wichtig; die volle Tragweite wird dann hoffentlich im fortgeschrittenen Teil ab Kapitel 43 deutlich.

Zeiger: Bei Zeigern und Adressen handelt es sich um einen sehr speziellen Datentyp, den wir erst in Kapiteln 45 und 46 kennen lernen werden. Aber auch diese beiden können Bestandteile von Ausdrücken sein.

22.2 Bool'sche Ausdrücke: `wahr` und `falsch`

Im Gegensatz zu den meisten anderen Programmiersprachen hat C keinen *eigenen* Datentyp für das Ergebnis logischer Operationen (Vergleiche, Verknüpfungen). Die Programmiersprache C macht es sich ganz einfach:

Regel 1 (`falsch`): *Alles* was null ist, ist `falsch`.

Regel 2 (`wahr`): *Alles* was ungleich null ist, ist `wahr`.

Regel 3: Ist das Ergebnis einer logsichen Operation (Vergleiche, Verknüpfungen) wahr, wird dieses durch eine `1` ersetzt; ist sie hingegen falsch, wird es durch eine `0` ersetzt.

Kapitel 23

Anweisungen, Blöcke und Klammern

Wir haben bereits einige Anweisungen kennengelernt. Dazu zählten Variablendefinitionen, Zuweisungen, Ein-/Ausgaben etc. Diese Anweisungen wurden immer mit einem Semikolon abgeschlossen. Nun gibt es in C aber noch ein paar Fallunterscheidungen und Schleifen. Diese erlauben aber nur eine einzige Anweisung. Um dort mehr als eine Anweisung unterzubringen, muss man diese mittels geschweifter Klammern `{}` zu einem Block zusammenfassen. Dieser Block gilt dann wie eine einzige Anweisung.

Ansonsten gilt, dass man auch eine einzige Anweisung zu einem Block zusammen fassen kann. Das wirkt auf den ersten Blick etwas überflüssig bzw. umständlich. Aber es hat den Vorteil, dass man später keine Klammern mehr setzen muss, wenn man doch noch eine weitere Anweisung benötigt. Dies erscheint auf den ersten Blick vielleicht ein wenig überflüssig, erleichtert später aber (dem Programmieranfänger) die Arbeit beim Finden und Beseitigen etwaiger Programmierfehler.

23.1 Verwendung

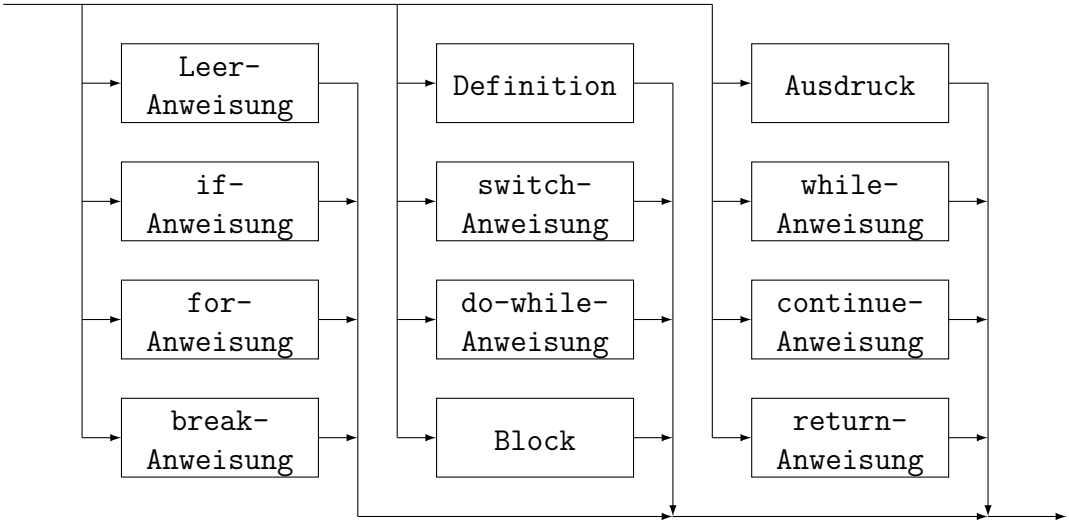
Vorab: Alle Präprozessor-Direktiven, also diejenigen Zeilen, die mit einem Doppelkreuz `#` anfangen, werden direkt vom Präprozessor bearbeitet und *entfernt*, sodass diese vom eigentlichen Compiler nicht mehr gesehen werden. Vergleiche hierzu auch Kapitel 38 und 39. Insofern gelten Präprozessor-Direktiven nicht als Anweisungen, obwohl sie wichtige Bestandteile eines jeden C-Programms sind.

Bisher konnten wir Variablen definieren und ihnen Werte zuweisen. In den Kapiteln 24 bis 28 kommen noch einige Kontrollstrukturen hinzu. Ferner benötigen wir in Funktionen eine `return;`-Anweisung auf die wir in Kapitel 44 näher eingehen. Die folgende Tabelle fasst die zwölf möglichen Anweisungen zusammen und illustriert diese mit jeweils einem kleinen Beispiel.

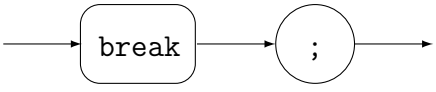
Anweisung	Beispiel
Leeranweisung	`; // nur ein Semikolon`
Definition	`int i, j;`
Ausdruck	`i * j + 3;`
	`i = 3;`
	`printf( "Hallo\n");`
Block	`{i = 1; j = 2; }`
if-Anweisung	`if (i == 1) j = 2; else j = 3;`
switch-Anweisung	`switch( i ){case 1: ... case n: }`
while-Anweisung	`while( i < 3 ) i = i + 1;`
for-Anweisung	`for( i = 0; i < 3; i = i +1 )`
do-while-Anweisung	`do i = i + 1; while( i < 3 );`
break-Anweisung	`break;`
continue-Anweisung	`continue;`
return-Anweisung	`return;`

23.2 Syntax

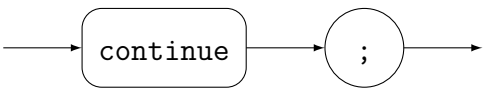
Anweisung



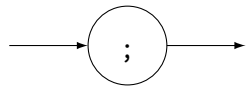
break-Anweisung



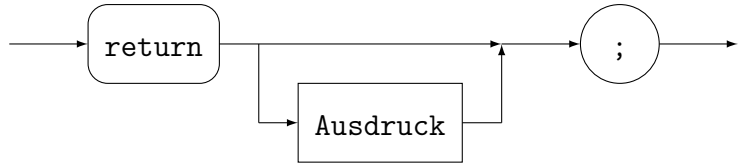
continue-Anweisung



Leer-Anweisung



return-Anweisung

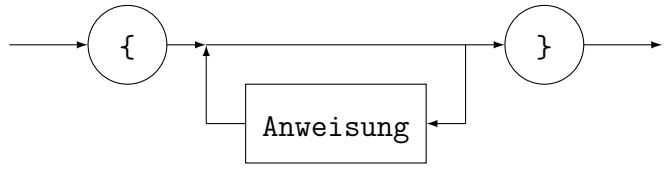


23.3 Der Anweisungs-Block

Wie bereits oben erwähnt, ist ein Block eine Aneinanderreihung einzelner Anweisungen, die mittels der geschweiften Klammern `{}` (zu einem einzigen Block) zusammengefasst werden, der syntaktisch wiederum wie eine einzelne Anweisung behandelt wird.

Syntaxdiagramm: Das Syntaxdiagramm dazu sieht folgendermaßen aus:

Block



Beispiele: Zum besseren Verständnis folgen hier zwei kleine Beispiele, die wiederum weitestgehend sinnfrei sind:

```
1 {
2     a = 2;
3     b = 3;
4     F = a * b;
5 }
```

```
1  // zuerst ein aeusserer block
2  {
3     a = 2; b = 3;
4     // und jetzt kommt noch ein innerer
5     {
6        innen = 3 * a/2;
7        innen = innen * b/2;
8     }
9     F = a * b - innen;
10 }
```

Hinweis: Bei der Verwendung von Anweisungsblöcken ist eines sehr wichtig: sie werden *nicht* mit einem Semikolon abgeschlossen, denn sonst wären es *zwei* eigenständige Anweisungen. Mit anderen Worten: Hinter der geschlossenen geschweiften Klammer kommt *kein* Semikolon. Zwischen den Klammern könnten auch beliebige andere Anweisungen stehen. Dies schließt auch weitere Blöcke ein.

Regeln für geschweifte Klammern: Die Regeln für die Verwendung geschweifter Klammern lassen sich wie folgt zusammenfassen:

1. Alle Anweisungen einer Funktion müssen mittels geschweifter Klammern zu einem Block zusammengefasst werden.
2. Alle Kontrollstrukturen (siehe die folgenden Kapitel 24 bis 28) erlauben nur eine einzige Anweisung; wird mehr als eine Anweisunge benötigt, müssen diese zu einem Block zusammengefasst werden.
3. Wenn man möchte, kann man auch nur eine einzige Anweisung in geschweifte Klammern setzen und damit zu einem Block zusammenfassen; das hat keine unmittelbaren Konsequenzen, vereinfacht aber *möglicherweise* die Programmänderung zu einem späteren Zeitpunkt.
4. Anweisungsblöcke erlauben etwas mehr „Fein-Tuning“ bezüglich der Sichtbarkeit von Variablen, worauf wir aber erst in Kapitel 56 eingehen.

Kapitel 24

Einfache Fallunterscheidung: `if-else`

Auf der Ebene der abstrakten Programmierung haben wir das Konzept der einfachen Fallunterscheidung bereits in Kapitel 13 besprochen. Das entsprechende Analogon in der Programmiersprache C heißt `if` und gilt als eine einzige Anweisung im Sinne der C-Syntax. Die `if`-Anweisung kommt in der Programmiersprache C in zwei Varianten vor: `if (Ausdruck) Anweisung_1; else Anweisung_2;` und ohne `else`-Teil: `if (Ausdruck) Anweisung_1;`.

24.1 Verwendung

C-Syntax

```
if ( Ausdruck )
   Anweisung_1;
else Anweisung_2;
```

Abstrakte Programmierung

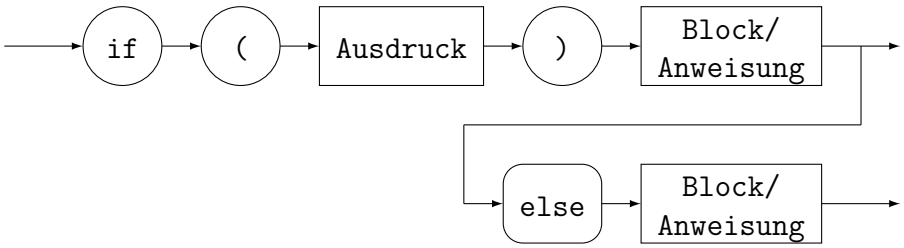
```
wenn  Bedingung
dann  Anweisung 1
sonst Anweisung 2
```

Die folgenden fünf Punkte sollten beachtet werden:

1. Der `else`-Teil kann komplett entfallen.
2. Jede der Anweisungen kann leer sein; also nur ein Semikolon `;`.
3. Bei mehr als einer Anweisung müssen diese mittels geschweifter Klammern `{}` zu einem Block zusammengefasst werden.
4. Die runden Klammern `()` sind Bestandteil der `if`-Anweisung, gehören also *nicht* zum Ausdruck und sind *unbedingt* notwendig.
5. Bei verschachtelten `if`-Anweisungen, gehört das `else` zum nächstgelegenen `if`.

24.2 Syntaxdiagramm

if-Anweisung



24.3 Korrekte Beispiele

Beispiel 1: Bestimme das Maximum `max` zweier Zahlen `a` und `b`.

```
1 if ( a > b )
2    max = a;
3 else max = b;
```

```
1 max = b;
2 if ( a > b )
3    max = a;
```

```
1 max = b;
2 if ( a > b )
3    max = a;
4 else ;
```

```
1 max = a;
2 if ( a > b )
3    ;
4 else max = b;
```

```
1 max = a;
2 if ( a > b )
3 {
4 }
5 else max = b;
```

```
1 if ( a > b )
2 {
3    max = a;
4 }
5 else max = b;
```

Alle sechs Varianten sind funktionell identisch und reine Geschmacksache.

Beispiel 2: Vorzeichenbestimmung einer Variablen `x`: `i = 1` für `x` größer null, `i = 0` für `x` gleich 0 und `i = -1` für `x` kleiner null:

```
1 if ( x > 0 )
2    i = 1;
3 else if ( x < 0 )
4         i = -1;
5      else i = 0;
```

```
1 if ( x >= 0 )
2    if ( x > 0 )
3       i = 1;
4    else i = 0;
5 else i = -1;
```

```
1 if ( x > 0 )
2    i = 1;
3 else {
4        if ( x < 0 )
5           i = -1;
6        else i = 0;
7      }
```

```
1 if ( x >= 0 )
2 {
3    if ( x > 0 )
4       i = 1;
5    else i = 0;
6 }
7 else i = -1;
```

Alle vier Beispiele sind funktionell identisch und die jeweils untereinander stehenden Beispiele unterscheiden sich nur in der Klammersetzung.

24.4 Fehlerhafte Beispiele

Die folgenden Beispiele sind inhaltlich völlig sinnlos. Sie dienen nur zum beispielhaften Aufzeigen möglicher Fehler im Quelltext.

```
1 // fehlende () Klammern beim if
2
3 if a < b
4    a = b;
5 else max = b;
```

```
1 // der dann-Teil fehlt
2
3 if ( a < b )
4 else max = b;
```

```
1 // fehlende {} Klammern
2 // im dann-Teil
3
4 if ( a < b )
5    max = a;
6    a = b;
7 else max = b;
```

```
1 // ein else-Teil
2 // zu viel
3
4 if ( a < b )
5    a = b;
6 else max = b;
7 else min = a;
```

Kapitel 25

Mehrfache Fallunterscheidung: `switch`

Für die mehrfache Fallunterscheidung gibt es in der Programmiersprache C die `switch`-Anweisung, innerhalb der mittels verschiedener `case-Label:`-Kombinationen die konkrete Auswahl getroffen wird. Die sich hinter einem `case-Label:` befindlichen Anweisungen *sollten* mit einem `break` abgeschlossen werden, da es sonst zu einem anderen Verhalten kommt. Bei den `Label`s innerhalb der `case`-Auswahl muss es sich um konstante, ganzzahlige Werte (`int`) handeln.

25.1 Verwendung

C-Syntax

```
switch( Ausdruck )
{
   case Wert_1: Anweisung_1;
                break;
   ...................
   ...................
   case Wert_n: Anweisung_n;
                break;
   default    : Anweisung_x;
                break;
}
```

Abstrakte Programmierung

```
auswahl: Ausdruck
   wenn Wert 1: Anweisung 1
   wenn Wert 2: Anweisung 2
   ........................
   wenn Wert n: Anweisung n
   sonst      : Anweisung x
```

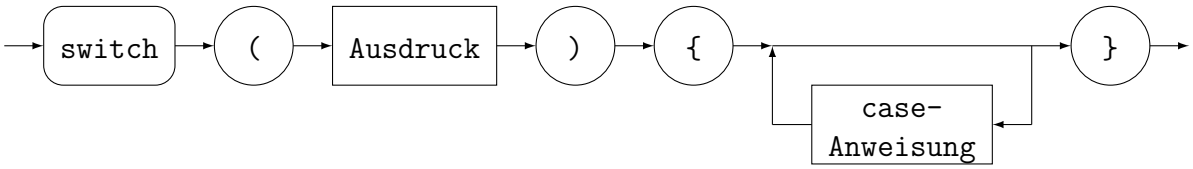
Hinweise: Die folgenden sechs Punkte sollten beachtet werden:

1. In diesem Beispiel steht `C_x` für eine ganzzahlige Konstante.
2. Sowohl die gezeigten runden `()` als auch die geschweiften `{}` Klammern gehören zur `switch`-Anweisung und müssen *in jedem Fall* gesetzt werden.

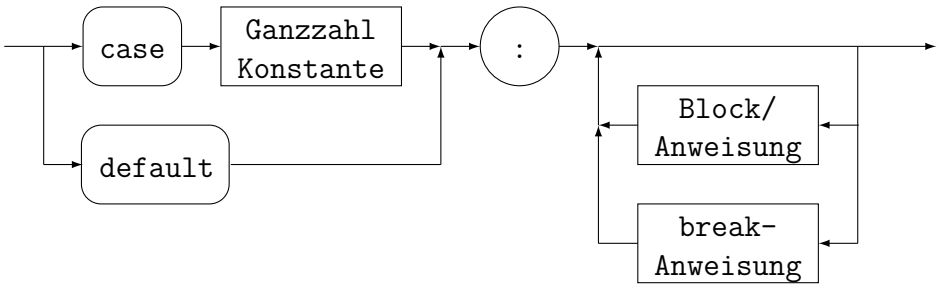
3. Hinter einer `case`-Auswahl können eine oder mehrere Anweisungen stehen. Eine weitere Klammerung mittels geschweifter `{}` Klammern ist nicht notwendig; die Anweisungsfolge wird erst durch ein `break` beendet.
4. Das Label `default:` kann an jeder beliebigen Stelle innerhalb der geschweiften `{}` Klammern stehen; muss also nicht notwendigerwesie zum Schluss kommen. Der Doppelpunkt `:` gehört zum Label `default` dazu.
5. Sollte kein `Label:` den Wert des `Ausdrucks` haben, wird mit den Anweisungen fortgefahren, die hinter dem Label `default:` stehen. Sollte auch dieses Label nicht vorhanden sein, wird keine Anweisung ausgeführt.
6. In einem Anweisungsteil können auch mehrere `break`-Anweisungen stehen, was aber häufig nicht sinnvoll ist.

25.2 Syntaxdiagramm

switch-Anweisung



case-Anweisung



Es sei hier nochmals darauf hingewiesen, dass hinter einem `case` immer *ein* einzelner Wert stehen muss, bei dem es sich um eine *ganzzahlige Konstante* handelt. Diese Konstanten müssen also *immer* vom Typ `int` sein oder sich darauf zurückführen lassen. Da der C-Compiler auch den Datentyp `char` (siehe Kapitel 30) als ganze Zahlen behandelt, sind hinter einem `case` auch *einzelne* Zeichen zulässig.

Hinter einem `case` sind Konstanten vom Typ `double` (siehe Kapitel 32) *nicht zulässig*. Eine Verarbeitung derartiger Ausdrücke und Werte muss also durch entsprechende Vergleichsoperationen innerhalb geeigneter `if`-Anweisungen durchgeführt werden.

25.3 Korrekte Beispiele

Auch hier folgen wieder einige Beispiele, die teilweise nicht wirklich sinnhaft sind, sondern nur die Verwendung der `switch`-Anweisung illustrieren sollen:

Beispiel 1: Stelle fest, ob `i` gleich 1, 2 oder 3 ist. Zwei funktionell identische Beispiele sehen wie folgt aus:

```
1  switch( i )
2  {
3    case 1: printf( "1\n" );
4            break;
5    case 2: printf( "2\n" );
6            break;
7    case 3: printf( "3\n" );
8            break;
9    default:printf("sonst\n");
10           break;
11 }
```

```
1  switch( i-1 )
2  {
3    case 0: printf( "1\n" );
4            break;
5    case 1: printf( "2\n" );
6            break;
7    case 2: printf( "3\n" );
8            break;
9    default:printf("sonst\n");
10           break;
11 }
```

Beispiel 2: Ist ein gegebenes Zeichen `ch` eine Ziffer, ein Rechenzeichen oder ein Vokal? Der Datentyp `char` kommt erst in Kapitel 30. Daher sei Folgendes vorweggenommen: Durch das Umschließen der einzelnen Zeichen mittels zweier Apostrophs, weiß der Compiler, dass keine Variablen sondern die entsprechenden Zeichen-Konstante gemeint sind.

```
1  switch( ch )
2  {
3     case '0': case '1': case '2': case '3': case '4': case '5':
4     case '6': case '7': case '8': case '9':
5               printf( "Es handelt sich um eine Ziffer\n" );
6               break;
7     case '+': case '-': case '/': case '*':
8               printf( "Es handelt sich um einen Operator\n" );
9               printf( "Bitte einen Operanden eingeben\n" );
10              break;
11    case 'a': case 'e': case 'i': case 'o': case 'u':
12    case 'A': case 'E': case 'I': case 'O': case 'U':
13              printf( "Es handelt sich um ein Vokal\n" );
14              break;
15 }
```

Dieses Beispiel zeigt, dass vor einem Anweisungsteil auch mehrere `case-Label:` kombinationen auftauchen können. Ferner zeigt die mittlere Auswahl, dass hinter dem Wert auch mehr als eine Anweisung stehen kann.

25.4 Fehlerhafte Beispiele

Zum Zeigen möglicher Fehler wird hier auf das erste Beispiel zurückgegriffen:

```
1 // hier fehlt )-Klammer zu
2
3 switch( i
4 {
5    case 1: printf( "1\n" );
6            break;
7    case 2: printf( "2\n" );
8            break;
9 }
```

```
1 // hier fehlt {-Klammer auf
2
3 switch( i )
4
5    case 1: printf( "1\n" );
6            break;
7    case 2: printf( "2\n" );
8            break;
9 }
```

```
1 // hier fehlt ein ':'
2
3 switch( i )
4 {
5    case 1  printf( "1\n" );
6            break;
7    case 2: printf( "2\n" );
8            break;
9 }
```

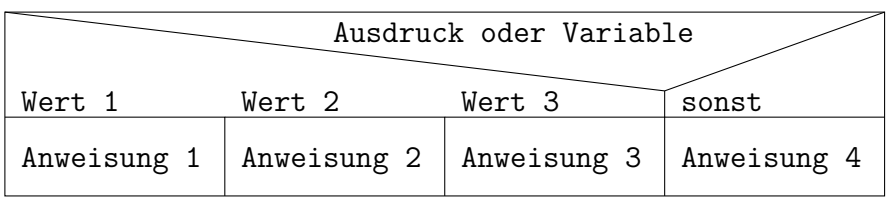
```
1 // hier fehlt ein 'case'
2
3 switch( i )
4 {
5         1: printf( "1\n" );
6            break;
7    case 2: printf( "2\n" );
8            break;
9 }
```

```
1 // hier fehlt ein ':'
2
3 switch( i )
4 {
5    case 1: printf( "1\n" );
6            break;
7    default printf( "2\n" );
8            break;
9 }
```

```
1 // variable i statt konst.
2
3 switch( i )
4 {
5    case 1: printf( "1\n" );
6            break;
7    case i: printf( "2\n" );
8            break;
9 }
```

25.5 Diskussion: `switch` versus Softwareengineering

Das von der Programmiersprache C angebotene `switch`-Konstrukt entspricht nicht so ganz der Idee des Software Engineerings, wie sie in Kapitel 13 erläutert wurde. Zur weiteren Diskussion ist hier das entsprechende Struktogramm nochmals wiedergegeben:



Erst durch die Verwendung der `break`-Anweisung verhält sich die `switch`-Anweisung wie gewünscht. Aber diese `break`-Anweisungen sind optional, wie obiges Syntaxdiagramm deutlich zeigt. Aber was passiert nun, wenn ein `break` fehlt? Ganz einfach, das Programm wird weiter abgearbeitet, egal, ob davor ein weiteres Label steht oder nicht. Ein Beispiel:

```
1 switch( i )
2 {
3     case 1: printf( "1\n" );
4             break;
5     case 2: printf( "2\n" );
6     case 3: printf( "3\n" );
7     case 4: printf( "4\n" );
8             break;
9 }
```

Da einzelne `break`-Anweisungen fehlen, wird folgendes ausgegeben:

i	1	2	3	4
Ausgabe	1	234	34	4

Ob dieses Verhalten gewünscht ist oder nicht, ist eine andere Frage. Auf jeden Fall widerspricht es dem Grundgedanken der Strukturierten Programmierung, nach dem die Auswahl genau einen Pfad auswählt; dennoch, es erlaubt einem routinierten Programmierer, sehr kompakte Programme zu schreiben. Die `break`-Anweisung ist eine Art *Notausgang*; Das eigentliche Verhalten der `switch`-Anweisung läßt sich vielleicht mittels folgendem Struktogramm veranschaulichen, wobei durch die gestrichelten Linien nur nach rechts oder unten weiter gemacht werden kann:

switch-Anweisung

Ausdruck	Wert 1	Anweisung 1	break
	Wert 2	Anweisung 2	break
	Wert 3	Anweisung 3	break
	sonst	Anweisung 4	break

Kapitel 26

Die `while`-Schleife

Bei der `while`-Schleife handelt es sich, wie bereits in Kapitel 14 besprochen, um eine *pre-checked loop*, die jedesmal vor Ausführung des Schleifenrumpfes die Schleifenbedingung überprüft. In der Programmiersprache C hat die `while`-Schleife fast das gleiche Aussehen, wie in der abstrakten Programmierung.

26.1 Verwendung

C-Syntax

```
while( Ausdruck )
   Anweisung;
```

Abstrakte Programmierung

```
solange Bedingung erfüllt
wiederhole Anweisung
```

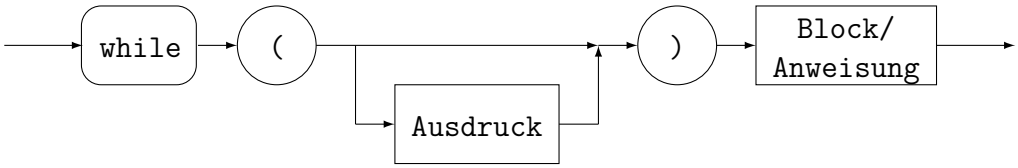
Hinweise: Die folgenden sechs Punkte sollten beachtet werden:

1. Im Schleifenrumpf darf nur eine Anweisung stehen. Wird mehr als eine Anweisung benötigt, sind diese mittels geschweifter Klammern `{}` zu einem Block zusammenzufasssen.
2. Die `while`-Schleife wird so lange durchlaufen, wie der `Ausdruck` wahr, d.h. ungleich null, ist.
3. Wird im Schleifenrumpf eine `break`-Anweisung ausgeführt, wird zur nächsten Anweisung *hinter* der Schleife gegangen.
4. Wird im Schleifenrumpf eine `continue`-Anweisung ausgeführt, wird an *das Ende* der Schleife gegangen und mit der Überprüfung der Schleifenbedingung weiter gemacht; es wird also ein neuer Schleifendurchlauf begonnen.
5. Wird im Schleifenrumpf eine `return;`-Anweisung ausgeführt, wird die gesamte Funktion verlassen.

6. Wird der leere Ausdruck verwendet, was erlaubt ist, wird dieser durch eine Konstante 1 ersetzt. Um eine Endlosschleife zu vermeiden, muss eine der Anweisungen aus den Punkten 3 oder 5 vorhanden sein.

26.2 Syntaxdiagramm

while-Schleife



26.3 Korrekte Beispiele

Beispiel 1: Berechne die Summe der Quadratzahlen von eins bis zehn $s = 1^2 + 2^2 + \cdots + 10^2 = \sum_{i=1}^{10} i^2$. Wie in der Beschreibung der Quadratsumme, nehmen wir zwei Variablen, den Laufindex `i` und eine Quadratsumme `qsum`. Zwei funktionell identische Beispiele sind:

```
1 // normaler Programmansatz
2
3 i = 1;
4 qsum = 0;
5 while( i < 11 )
6 {
7     qsum = qsum + i * i;
8     i = i + 1;
9 }
```

```
1 // eher unschoen programmiert
2
3 i = 1; qsum = 0;
4 while( 1 )
5 {
6     qsum = qsum+i*i; i = i+1;
7     if ( i == 10 )
8         break;
9 }
```

Beispiel 2: Berechne die Fakultät $9! = 1 \cdot 2 \cdot \ldots \cdot 9$. Wie im vorherigen Beispiel benötigen wir zwei Variablen, einen Laufindex `i` und die Fakultät `f`. Ein einfaches C-Programm sieht wie folgt aus:

```
1 i = 2; f = 1;
2 while( i < 10 )
3 {
4     f = f * i;
5     i = i + 1;
6 }
```

26.4 Fehlerhafte Beispiele

Ein paar der möglichen Fehlerquellen (in sinnfreien Programmstücken) sehen wie folgt aus:

```
1 // hier fehlt )-Klammer zu
2 // merkt der Compiler
3
4 while( i < 20
5 {
6    j = j + 5 + 1;
7    i = i + 2;
8 }
```

```
1 // hier fehlt {-Klammer auf
2 // merkt der Compiler nicht!
3
4 while( i < 20 )
5
6    j = j + 5 + 1;
7    i = i + 2;
8 }
```

```
1 // Endlosschleife, da i
2 // nicht erhoeht wird.
3 // merkt der Compiler nicht!
4
5 while( i < 20 )
6 {
7    j = j + 5 + 1;
8 }
```

```
1 // Endlosschleife, da
2 // Semikolon ; vor Block {}
3 // merkt der Compiler nicht!
4
5 while( i < 20 );
6 {
7    j = j + 5 + 1;
8 }
```

26.5 Diskussion: `break` in `while`-Schleifen

Die `break`-Anweisung erlaubt es dem erfahrenen Programmierer, sehr kompakten Code zu schreiben, da man auf „umständliche“ Abfragen und Bedingungen verzichten kann. Dennoch läuft – wie bei der `switch`-Anweisung – die Verwendung der `break`-Anweisung dem Grundgedanken der Strukturierten Programmierung entgegen: Dieser besagt, dass der Schleifenrumpf am Ende verlassen wird und nicht einfach mittendrin. Insofern sollte die `break`-Anweisung mit Sorgfalt verwendet werden.

Kapitel 27

Die `for`-Schleife

Gemäß Kapitel 14 ist die Zählschleife die zweite Form der *pre-checked loop*. Die Programmiersprache C bietet mit der `for`-Schleife ein entsprechendes Pendant. Aus didaktischer Sicht liegt nun ein Problem darin, dass die `for`-Schleife keine echte Zähl- sondern eher eine verallgemeinerte `while`-Schleife ist, weshalb wir sie in einigen Zwischenschritten einführen.

27.1 Verwendung: ein erster Ansatz

C-Syntax

```
for( A_1 ; A_2 ; A_3 )
   Anweisung;
```

Abstrakte Programmierung

```
für x = a bis e schrittweite s
wiederhole Anweisung;
```

In obiger Darstellung sieht man, dass die Schleifenbedingung aus drei Ausdrücken `A_1`, `A_2` und `A_3` besteht, die jeweils mit einem Semikolon voneinander getrennt sind. Die Verwendung von Semikolons innerhalb einer Schleifenbedingung ist neu und mutet ggf. etwas sonderbar an. Sie sind hier aber notwendig, wie wir in Abschnitt 27.3 noch sehen werden. Nun magst du aber erstaunt sagen: *„Ausdrücke? Muss da nicht eine Zählvariable stehen, die auf einen Anfangswert gesetzt und später erhöht und gegenüber einem Grenzwert überprüft wird?“* Nicht schlecht bemerkt! Stimmt! Aber jetzt erinnern wir uns an eine etwas subtile Bemerkung aus Kapitel 22, nach der auch eine Zuweisung ein Ausdruck darstellt. Anmerkung: Im engeren Sinne, besteht die Schleifenbedingung nur aus dem Ausdruck `A_2`, da sein Wert über das Abbrechen oder Weiterlaufen der Schleife entscheidet.

Beispiel: Um die Diskussion etwas konkreter am Beispiel zu führen, drucken wir einfach mal die Zahlen von 1 bis 10. Nicht besonders komplex, aber der Sache sehr dienlich:

```
1 for( i = 1; i < 11; i = i + 1 )
2     printf( "i= %d\n", i );
```

mit den drei Ausdrücken: `A_1: i = 1`, `A_2: i < 11`, `A_3: i = i + 1`

Funktionsweise: Die Funktionsweise unseres kleinen Beispiels ist wie das Einführungsbeispiel aus Kapitel 14 erwarten läßt: Zuerst wird die Zählvariable `i=1` auf den Wert 1 gesetzt. Anschließend wird überprüft `i < 11` (*pre-checked loop*), ob die Schleifenbedingung erfüllt ist. Da dies in diesem Fall erfüllt ist, wird der Schleifenrumpf `printf( "i= %d\n", i );` einmal durchlaufen. Anschließend wird auf jeden Fall auch der dritte Ausdruck `A_3: i = i + 1` ausgeführt. Jetzt wird wieder wie am Anfang die Schleifenbedingung `i < 11` überprüft. Dieser Zyklus wird so lange fortgesetzt, bis die Schleifenbedingung `i < 11` nicht mehr erfüllt ist, was in unserem Fall bei `i == 11` der Fall sein wird.

Die `for`-Schleife als generalisierte `while`-Schleife: Jetzt spulen wir geistig nochmals zurück an den Anfang dieses Abschnitts. Für den Fall, dass sich *keine* `continue`-Anweisung im Schleifenrumpf befindet, können wir die `for`-Schleife wie folgt in eine `while`-Schleife umwandeln:

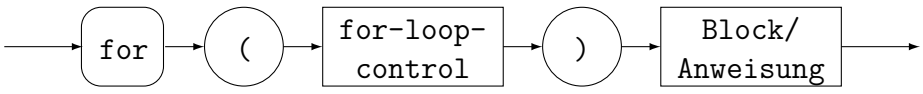
```
1 // fuer for-schleifen ohne continue;-Anweisung gilt folgendes:
2
3 for( A_1 ; A_2 ; A_3 )          A_1;
4    Anweisung ;                  while( A_2 )
5                                 {
6                                    Anweisung;
7                                    A_3;
8                                 }
```

Hinweise: Die folgenden zwei Punkte sollten beachtet werden:

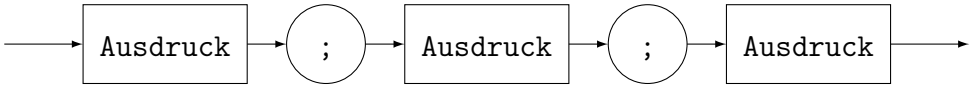
1. Da es sich bei der `for`-Schleife nahezu (Ausnahme `continue`-Anweisung) um eine generalisierte `while`-Schleife handelt, gelten alle Hinweise bezüglich der Zahl der Anweisungen im Schleifenrumpf und der Bündelung in Form eines Blockes, dem möglichen Wegfall des Ausdrucks `A_2` sowie der `break`- und `return`-Anweisungen.
2. Wird im Schleifenrumpf eine `continue`-Anweisung ausgeführt, wird an *das Ende* der Schleife gegangen und mit der Abarbeitung von Ausdruck `A_3` fortgefahren. Erst *anschließend* erfolgt die erneute Überprüfung der Schleifenbedingung.

27.2 Syntaxdiagramm

`for-Schleife`



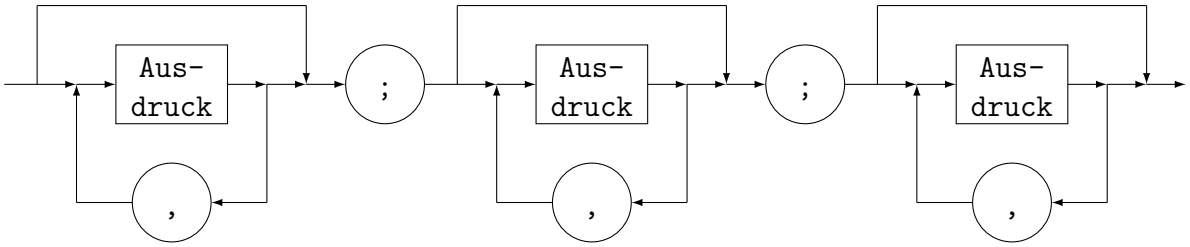
`for-loop-control (vereinfacht)`



27.3 Verwendung

Nun müssen wir uns noch an eine weitere subtile Bemerkung erinnern, die wir in Kapitel 22 gemacht haben: Auch die Aneinanderreihung mehrerer Ausdrücke gilt wieder als *ein* Ausdruck. Das heißt, wir könnten beispielsweise im Ausdruck `A_1` auch mehrere Initialisierungen vornehmen, beispielsweise `i = 1, sum = 0`. In Form eines Syntaxdiagramms wäre dies:

`for-loop-control (vollständig)`



Angemerkt sei, dass dies nicht nur bei `for`-Schleifen gilt, sondern *überall*, wo man Ausdrücke verwendet. Nur eben ist dieses Verständnis bei der `for`-Schleife besonders wichtig. Anhand des erweiterten Syntaxdiagramms wird hoffentlich auch klar, weshalb zwischen den drei Ausdrücken `A_1`, `A_2` und `A_3` jeweils ein Semikolon und kein Komma stehen muss: wäre es anders, könnte der Compiler die Ausdrücke den drei Teilen nicht mehr klar zuordnen.

27.4 Korrekte Beispiele

Beispiel 1: Wir greifen hier nochmals das eingangs besprochene Beispiel auf, die Zahlen von 1 bis 10 auszugeben. Den Quellcode haben wir bereits in Abschnitt 27.1 besprochen. Hier folgen zwei weitere Varianten, die alle miteinander funktionell identisch sind:

```
1 // drucke die zahlen 1..10
2
3 i = 1;
4 for( ; i < 11; i = i + 1 )
5     printf( "%d\n", i );
```

```
1 i = 1;
2 for( ; i < 11; )
3 {
4     printf( "%d\n", i );
5     i = i + 1;
6 }
```

Beispiel 2: Als Beispiel greifen wir wieder die Berechnung der Quadratsumme aus Kapitel 26 auf: Berechne die Summe der Quadratzahlen von eins bis zehn $s = 1^2+2^2+\cdots+10^2 = \sum_{i=1}^{10} i^2$. Wir nehmen wieder zwei Variablen vom Typ `int`, den Laufindex `i` und die Quadratsumme `sum`. Die folgenden drei Implementierungen sehen unterschiedlich aus, sind funktional identisch:

```
1 // quadratsumme von 1..10
2
3 sum = 0;
4 for( i=1; i<11; i = i+1 )
5    sum = sum + i * i;
```

```
1 // quadratsumme von 1..10
2
3 for( sum=0, i=1; i<11; i=i+1 )
4    sum = sum + i * i;
```

```
1 // quadratsumme von 1..10
2
3 for( sum = 0, i = 1; i < 11; i = i + 1, sum = sum + i * i )
4    ;
```

Alle drei Beispiele sind funktional (und eigentlich auch implementierungstechnisch) völlig identisch. Dies kann man gut sehen, wenn man jedes der drei Beispiele in eine entsprechende `while`-Schleife umwandelt, was eine gute Übung für jeden Leser ist :-) !

27.5 Fehlerhafte Beispiele

In obigen Programmstücken könnten beispielsweise folgende Fehler gemacht werden:

```
1 // hier fehlt ein Semikolon ;
2 // innerhalb der () Klammern
3
4 psum = 1; i = 1;
5 for( i < 11; i++ )
6    psum = psum * i;
```

```
1 // hier fehlt eine runde
2 // Klammer (
3
4 psum = 1; i = 1;
5 for ; i < 11; i++ )
6    psum = psum * i;
```

```
1 // ein Semikolon ; zu viel in der Schleifenbedingung()
2
3 for( p = 1; i = 1; i < 11; i++, p = p * i )
4    ;
```

27.6 Diskussion: `break` in `for`-Schleifen

Hier gilt die selbe Diskussion, wie sie bereits in Kapitel 26 geführt wurde: `break` und `continue` widerstreben dem Grundgedanken der Strukturierten Programmierung und sollten mit Bedacht eingesetzt werden.

Kapitel 28

Die `do-while`-Schleife

Bei der `do-while`-Schleife handelt es sich, wie bereits in Kapitel 14 besprochen, um eine *post-checked loop*, die jedesmal *nach* Ausführung des Schleifenrumpfes die Schleifenbedingung überprüft. In der Programmiersprache C hat diese *post-checked loop* die Form „solange–wie“ und *nicht* „solange–bis“. Mit dieser kleinen Ausnahme ist sie syntaktisch fast genau so aufgebaut wie die `while`-Schleife.

28.1 Verwendung

`C-Syntax`

```
do Anweisung;
while( Ausdruck );
```

`Abstrakte Programmierung`

```
wiederhole Anweisung
solange Bedingung erfüllt
```

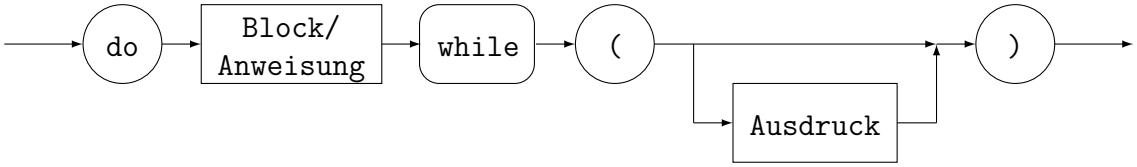
Hinweise: Die folgenden sieben Punkte sollten beachtet werden:

1. Im Unterschied zur regulären `while`-Schleife steht hinter der Schleifenbedingung auch ein Semikolon `;`. In einer regulären `while`-Schleife würde dies häufig zu Endlosschleifen führen, hier ist das Semikolon notwendig. Dies kann man sich leicht merken: Die `do-while`-Anweisung ist syntaktisch erst hinter der Bedingung zu Ende und das Ende einer Anweisung/Schleife wird mittels eines Semikolon gekennzeichnet.
2. Im Schleifenrumpf darf nur eine Anweisung stehen. Wird mehr als eine Anweisung benötigt, sind diese mittels geschweifter Klammern `{}` zu einem Block zusammenzufasssen.
3. Die `do-while`-Schleife wird so lange durchlaufen, wie der `Ausdruck` wahr, d.h. ungleich null, ist.
4. Wird im Schleifenrumpf eine `break`-Anweisung ausgeführt, wird zur nächsten Anweisung *hinter* der Schleife gegangen.

5. Wird im Schleifenrumpf eine `continue`-Anweisung ausgeführt, wird an *das Ende* der Schleife gegangen und mit der Überprüfung der Schleifenbedingung weiter gemacht.
6. Wird im Schleifenrumpf eine `return`-Anweisung ausgeführt, wird die gerade ausgeführte Funktion verlassen.
7. Wird der leere Ausdruck verwendet, was erlaubt ist, wird dieser durch eine Konstante 1 ersetzt. Um eine Endlosschleife zu vermeiden, muss eine der Anweisungen aus den Punkten 4 oder 6 vorhanden sein.

28.2 Syntaxdiagramm

do-while-Schleife



28.3 Korrekte Beispiele

Beispiel 1: Berechne das Produkt der Zahlen von eins bis zehn $p = 1 \times 2 \times \cdots \times 10 = \prod_{i=1}^{10} i$. Wie in der Formel des Produktes, nehmen wir zwei Variablen, den Laufindex `i` und ein Produkt `prod`:

```
1  i = 1;
2  prod = 1;
3  do {
4        prod = prod * i;
5        i = i + 1;
6     }
7  while( i < 11 );
```

Beispiel 2: Berechne die Fakultät $9! = 1 \cdot 2 \cdot \ldots \cdot 9$. Wie im vorherigen Beispiel benötigen wir zwei Variablen, einen Laufindex `i` und die Fakultät `f`. Eine einfaches C-Programm sieht wie folgt aus:

```
1  i = 1; f = 1;
2  do {
3        f = f * i;
4        i = i + 1;
5     } while( i < 10 );
```

28.4 Fehlerhafte Beispiele

Ein paar der möglichen Fehlerquellen (in sinnfreien Programmstücken) sehen wie folgt aus:

```
1 // hier fehlt )-Klammer zu
2 // merkt der Compiler
3
4 do {
5       j = j + 5 + 1;
6       i = i + 2;
7     }
8 while( i < 20 ;
```

```
1 // hier fehlt {-Klammer auf
2 // merkt der Compiler nicht
3
4 do
5       j = j + 5 + 1;
6       i = i + 2;
7     }
8 while( i < 20 );
```

```
1 // Endlosschleife, da i
2 // nicht erhoeht wird.
3 // merkt der Compiler nicht
4
5 do {
6       j = j + 5 + 1;
7     }
8 while( i < 20 );
```

```
1 // Semikolen ; vor Block {}
2 // merkt der Compiler!
3
4 do ;
5     {
6       j = j + 5 + 1;
7     }
8 while( i < 20 );
```

28.5 Diskussion: `break` in Schleifen

Die `break`-Anweisung erlaubt es dem erfahrenen Programmierer, sehr kompakten Code zu schreiben, da man auf „umständliche“ Abfragen und Bedingungen verzichten kann. Dennoch läuft auch hier wie bei der `while`-Anweisung die Verwendung der `break`-Anweisung dem Grundgedanken der Strukturierten Programmierung entgegen: Dieser besagt, dass der Schleifenrumpf am Ende verlassen wird und nicht einfach mitten drin. Insofern sollte die `break`-Anweisung mit Sorgfalt verwendet werden.

Kapitel 29

Die ASCII-Tabelle: die gängigste Kodierung von Zeichen im Rechner

Neben „langweiligen“ Zahlen kann man in der Programmiersprache C auch Zeichen und ganze Texte verarbeiten. Beides erfordert aber das Verständnis der Kodierung eines Zeichens. Im Regelfall wird dies auf heutigen Rechnern mittels der ASCII-Tabelle geregelt, die für jedes *US amerikanische* Zeichen einen eigenen Code (also einen eigenen Wert) definiert. Obwohl man als Programmierer nicht direkt mit der ASCII-Tabelle zu tun hat, ist sie dennoch *die* Grundlage der Zeichenrepräsentation innerhalb eines C-Programms. Daher ist auch das Verständnis der grundlegenden Struktur dieser Tabelle eine wesentliche Voraussetzung für die spätere Be- und Verarbeitung von Zeichen und Texten.

29.1 Die ASCII-Tabelle und ihre Eigenschaften

Wie bereits in Kapitel 5 erläutert wurde, besteht der Arbeitsspeicher aus einer riesigen Aneinanderreihung einzelner Bytes, die jeweils aus acht Bits bestehen, die ihrerseits die Werte „logisch Null“ und „logisch Eins“ annehmen können. Für die interne Kodierung von Zeichen wurde im Jahre 1963 [6] der *American Standard Code for Information Interchange* in Form einer Tabelle definiert, die wir oben auf der nächsten Seite reproduziert haben (Original siehe [6]).

Die ASCII-Tabelle hat folgende Eigenschaften:

1. Die Tabelle besteht aus lediglich 128 Zeichen, die Werte variieren von hexadezimal 0x00 bis 0x7F, was dezimal ausgedrückt die Zahlen von 0 bis 127 sind. Hierfür werden nur sieben Bit benötigt, sodass das achte Bit für andere Zwecke, beispielsweise für das Erkennen von Übertragungsfehlern, genutzt werden kann. Siehe hierzu auch [2].

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1...	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2...	*SP*	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*DEL*

2. Die ersten 32 Zeichen (0x00 bis 0x1F bzw. 0 bis 31) sind Sonderzeichen, die früher meist nur für die Realisierung von Übertragungsprotokollen verwendet wurden.

3. Einige der Sonderzeichen dienen der Formatierung der Ausgabe. Beispiele hierfür sind der horizontale Tabulator *HT* (0x9 bzw. 9) und der Wagenrücklauf *CR* (0xD bzw. 13). Der Tabulator sorgt dafür, dass die Ausgabe zur nächsten Tabulator Position (engl. *tab stop*) vor geht. Diese Tabulatorpositionen sind in der Regel alle acht Zeichen. Durch den Wagenrücklauf wechselt die Ausgabe auf den Anfang der nächsten Zeile. Beide Sonderzeichen haben ihren Ursprung in der *sehr alten* mechanischen Schreibmaschine.

4. Die nächsten 95 Zeichen (0x20 bis 0x7E bzw. 32 bis 126) sind die aus dem US-amerikanischen Zeichenvorrat bekannten druckbaren Zeichen.

5. Die 95 druckbaren Zeichen sind nicht wahllos durcheinander gewürfelt, sondern sinnvoll gruppiert. Beispielsweise erkennt man drei zusammenhängende Blöcke, bestehend aus den Ziffern `'0'-'9'`, den Großbuchstaben `'A'-'Z'` sowie den Kleinbuchstaben `'a'-'z'`. Innerhalb jeder Gruppe sind die Zeichen so angeordnet, wie wir es von unseren Zahlen bzw. unserem Alphabet gewohnt sind. Auch die drei Gruppen sind untereinander so sortiert, wie wir es gewohnt sind: die Ziffern vor den Großbuchstaben und diese vor den Kleinbuchstaben.

6. Es sei noch angemerkt, dass das Zeichen *SP* (0x20 bzw. 32) für das Leerzeichen (den nichtbedruckten Zwischenraum) steht.

7. Das letzte Zeichen *DEL* (0x7F bzw. 127) ist wieder ein Sonder- bzw. Steuerzeichen.

29.2 Erweiterungen der ASCII-Tabelle

Die ASCII-Tabelle wurde ursprünglich nur für die US-amerikanischen Bedürfnisse definiert. Entsprechend fehlen Umlaute wie ä, Ä, ü, ã, ø, á, â usw. Da die untere Hälfte (Kodierung von 0x80 bis 0xFF) der original ASCII-Tabelle noch frei war, wurden viele der im westlichen

Sprachgebrauch verbreiteten Sonderzeichen dort definiert und als erweiterte ASCII-Tabelle bezeichnet.

29.3 Unicode: der Zeichensatz für alle Sprachen

Aber auch die erweiterte ASCII-Tabelle ist sehr eingeschränkt, da in ihr nur die Zeichenkombinationen der westlichen Sprachen aufgenommen wurden. Und egal, wie man es anstellt, für die Zeichen anderer Sprachen wie Kyrillisch, Japanisch, Chinesisch, Hebräisch etc. ist kein Platz. Um hier eine „endgültige“ Lösung zu finden, wurde im Jahre 1991 der Unicode definiert [11], der für jedes Zeichen 16 Bits verwendet und somit die Kodierung aller auf der Erde bekannten Schriftzeichen erlaubt. Allerdings ist der Unicode *nicht* Bestandteil der Programmiersprache C. Aber in den Standardbibliotheken werden mittlerweile Funktionen angeboten, die die Verarbeitung dieser Zeichen recht einfach erlauben.

29.4 Proprietäre Zeichensätze

Es gibt noch weitere Zeichensätze, so beispielsweise der EBCDIC Zeichensatz von IBM. Dieser Zeichensatz wurde früher auf IBMs Großrechnern verwendet. Denkbar ist auch jede Form von Zeichensatz.

29.5 Schlussfolgerungen

Auch wenn in diesem Kapitel neben dem ASCII-Zeichensatz mehrere andere Zeichensätze kurz angesprochen wurden, so war der ASCII-Zeichensatz derjenige der eigentlich von allen Rechnern und Hardwaresystemen verwendet wurde.

Kapitel 30

Datentyp `char`: ein einzelnes Zeichen

Neben den Zahlen gehören einzelne Zeichen zu den grundlegenden einfachen Datentypen jeder Programmiersprache. Mittels des Datentyps `char` kann man *einzelne* Zeichen verarbeiten, die man ihrerseits in einfache Apostrophe `'` einschließt. Beispiele sind: `'a'`, `'x'`, `';'` und `'0'`. Damit kann man einfache Zeichen verarbeiten, ja/nein-Abfragen implementieren, Zahlen einlesen und vieles mehr machen. Einzelne Zeichen sind ein wesentlicher Baustein in Richtung Zeichenketten, die später in Kapitel 50 behandelt werden.

30.1 Verwendung

C-Syntax

```
char c, h;
c = 'a';
h = c + 1;
```

Abstrakte Programmierung

```
Variablen: Typ Zeichen: c, h
setze c = Zeichen a
setze h = Nachfolger von c
```

Hinweise: Bei Verwendung von `char`-Variablen und `char`-Zeichen (Konstanten) sollte folgendes beachtet werden:

1. Bevor eine Variable benutzt wird, muss sie definiert[1] werden. Dies passiert in obigem Beispiel in der ersten Zeile.
2. Eine Variable vom Typ `char` kann *nur ein* Zeichen aufnehmen; den Wert „kein Zeichen“ gibt es nicht.
3. Ein einzelnes Zeichen wird in der Regel zwischen zwei einzelne Apostrophe gesetzt. Für das, was zwischen die beiden Apostrophs kommt, gibt es die folgenden drei Möglichkeiten:

[1] Es geht auch noch ein klein wenig anders, aber das besprechen wir erst in Kapitel 55.

„Normale“-Zeichen: Ein einzelnes Zeichen wird in der Regel in Apostrophs eingeklammert. Beispielsweise entspricht `'a'` dem Zeichen *kleines a*.

Escape Sequenzen: Einige Sonderzeichen können mittels eines Backslashs \ sowie eines weiteren Zeichens kodiert werden:

Escape-Sequenz	Bedeutung
`'\t'`	Tabulator
`'\n'`	Zeilenwechsel
`'\r'`	Zeilenanfang
`'\''`	Apostroph
`'\0'`	Null-Byte (Zeichen mit der Kodierung 0)

Weitere Escape Sequenzen lassen sich in der Literatur finden, z.B. [7].

Direkte Oktal-Kodierung: Für den Fall, dass man den Zeichen-Code seines Rechners kennt, kann man die Zeichen auch mittels des Codes direkt angeben: Nach einem Backslash \ können bis zu drei oktale Ziffern (0..7) folgen. Beispiel `'\12'` kodiert dasjenige Zeichen, das in der ASCII-Tabelle den Dezimalwert zehn hat.

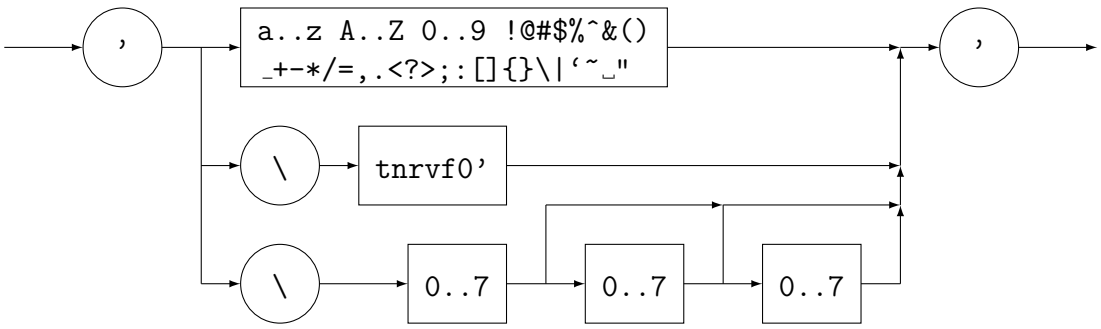
Diese Vorgehensweise ist aber im Allgemeinen nur in Spezialfällen zu empfehlen, da dadurch ein Programm in der Regel nicht mehr portierbar ist. Auch die Lesbarkeit und Änderungsfreundlichkeit leiden sehr darunter.

In allen drei Fällen gilt, dass die beiden Apostrophs nicht zum Zeichen gehören; sie zeigen dem Compiler nur an, dass der folgende Wert ein Zeichen, also ein Wert vom Datentyp `char`, ist.

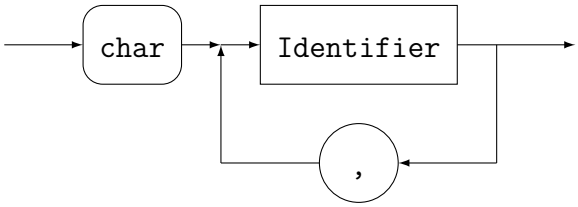
30.2 Syntaxdiagramme (vereinfacht)

Konstanten vom Typ `char` können wie folgt gebildet werden (wie Variablen vom Typ `char` definiert werden, zeigt das Syntaxdiagramm auf der folgenden Seite):

`char-Konstante`



`char-Definition (vereinfacht)`



30.3 Korrekte Beispiele

```
1 c = 'a';          // kleines a
2 c = '\n';         // Zeilenumbruch
3 c = '\60';        // nach ASCII-Tabelle Ziffer '0'
4 c = '\060';       // nach ASCII-Tabelle Ziffer '0'
5 c = '"';          // ein Anfuehrungszeichen
6 c = ';';          // ein Semikolon
7 c = '\'';         // ein Apostroph
```

30.4 Fehlerhafte Beispiele

```
1 c = "a';          // Anfuehrungszeichen statt Apostroph
2 c = 'xn';         // zwei statt einem Zeichen
3 c = '\80';        // keine oktale Zahl
4 c = 'x;           // zweiter Apostroph fehlt
```

30.5 Ausgabe eines `char` (Zeichens)

Auch einzelne Zeichen können mittels `printf()` ausgegeben werden. Hierfür gibt es die Formatierung `%c`. Beispiel: Mittels `printf( "Zeichen: %c\n", 'a' );` wird ausgegeben: `Zeichen: a`, wie zu erwarten war.

30.6 Einlesen von `char`-Werten

`char`-Werte kann man mittels der Funktion `scanf()` einlesen, sofern man als Formatierung `%c` angibt. Beispiel: `scanf( "%c", & c )`, wobei `c` eine Variable vom Typ `char` ist.

30.7 Definition einschließlich Initialisierung

Hier gilt das gleiche wie bereits in Kapitel 21.8 gesagt: Variablen vom Typ `char` können bereits bei ihrer Definition initialisiert werden. Beispielsweise `char c = 'a';`.

30.8 Interne Repräsentation

Die interne Repräsentation ist schnell erklärt: Jedes Zeichen belegt ein Byte, zumindest auf den meisten der zur Zeit verfügbaren PCs und Mikrocontroller. Die meist zugrundelie-

gende ASCII-Kodierung haben wir bereits in Kapitel 29 besprochen. Die byteweise interne Repräsentation bedeutet ferner, dass der Aufruf von `sizeof( char )` *meistens* den Wert 1 liefert; dies ist sogar im C-Standard [13] so festgelegt. Somit ist ein `char` auch die kleinste eigenständige Einheit, die mit einer Speicherzelle übereinstimmt. Zusammengefasst: Die Verwendung von Variablen und Werten vom Typ `char` ist eigentlich recht easy.

30.9 Rechenoperationen

Da der Datentyp `char` intern auf den Datentyp `int` abgebildet wird, sind prinzipiell alle vier Grundrechenarten erlaubt. Im Regelfall sind aber nur `+` und `-` sinnvoll. Beipielsweise kann man ein großes Zeichen `c` wie folgt in ein kleines umwandeln: `c = c - 'A' + 'a'`. Warum klappt das? Ganz einfach: Der erste Term `c - 'A'` „berechnet“, um das wievielte Zeichen es sich innerhalb den Bereiches `'A'` bis `'Z'` handelt. Durch die Addition von `+ 'a'` wird das kleine `'a'` um entsprechend viele Stellen verschoben. Wer es nicht glaubt, der probiere dies anhand der ASCII-Tabelle (Kapitel 29) einfach mal selbst aus.

30.10 Programmbeispiele

Bei der Verarbeitung von Zeichen kommt bei Programmieranfängern immer wieder die Frage auf: „Welchen ASCII-Code hat das Zeichen X?“ Aber diesen ASCII-Code brauchen wir gar nicht zu wissen, denn der Compiler weiss ihn. Würden wir ihn dennoch direkt verwenden, könnte es auf Zielrechnern, die *keinen* ASCII-Code verwendet, zu Problemen und ungeplanten Programmabläufen führen. Dies kommt zur Zeit nicht besonders oft vor, aber es passiert. Mit anderen Worten: Man nehme keine ASCII-Codes sondern die Zeichen und überlasse es dem Compiler, die Konvertierung vorzunehmen. Hier ein paar Beispiele.

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4      {
5          char c;
6          printf( "Soll weiter gemacht werden (j/n)? " );
7          scanf( "%c", & c );
8          if ( c == 'j' )
9             printf( "ok, mache weiter\n" );
10         else printf( "tschuess\n" );
11     }
```

Dieses Programm ist nicht besonders sinnvoll, sondern dient nur der Veranschaulichung. In Zeile 7 wird ein Zeichen eingelesen und in Zeile 8 auf seinen Wert überprüft. Je nach Eingabe gibt es eine entsprechende Ausgabe.

Das nächste Programmstück überprüft, ob eine Zeichenvariable einen Buchstaben enthält:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5         char c;
 6         printf( "Bitte ein Zeichen eingeben: " );
 7         scanf( "%c", & c );
 8         if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
 9            printf( "Das Zeichen ist ein Buchstabe\n" );
10         else printf( "Das Zeichen ist *kein* Buchstabe\n");
11     }
```

Die entscheidende Abfrage ist in Zeile 8. Diese Abfrage macht sich zunutze, dass die Buchstaben innerhalb der ASCII-Tabelle sortiert sind, wobei das `'a'` vor dem `'b'` kommt, das `'b'` vor dem `'c'` kommt usw. Aus der Schule sollte klar sein, dass ein Buchstabe entweder zwischen `'a'` und `'z'` *oder* `'A'` und `'Z'` liegen muss. Zur Abfrage noch folgender Hinweis: die inneren Klammern hätten weggelassen werden können, da in der Programmiersprache C Punktrechnung (logisches und) vor Strichrechnung (logisches oder) geht; in anderen Programmiersprachen wie beispielsweise Pascal ist dies nicht unbedingt der Fall.

Geradezu ein Klassiker ist die Umwandlung von Ziffern in die entsprechende `int`-Zahl:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4     {
5         char c = '7';
6         int i = c - '0';
7         printf( "Der Zahlenwert von %c ist %d\n", c, i );
8     }
```

Und tatsächlich, es wird 7 ausgegeben, denn der Compiler weiß ja, welchen ASCII-Wert die Ziffer `'0'` hat und subtrahiert diesen vom ASCII-Wert der Ziffer `'7'`.

30.11 Akademischer Hintergrund: `char` ist nicht `int`

Hier kommen wir zu einem etwas schwierigen Thema, da die Zusammenhänge in der Progammiersprache C doch etwas subtiler sind. Andere Programmiersprachen wie Pascal und Modula-2 sind diesbezüglich einfacher zu verstehen, denn diese Sprachen trennen sehr deutlich und bewusst zwischen den einzelnen Datentypen und verlangen eine sehr explizite und bewusste Umwandlung zwischen ihnen. In der Programmiersprache C ist das alles anders: der C-Compiler „denkt“ sich seinen Teil und versucht den Programmierer durch implizite Konvertierungen zu unterstützen. Dadurch kommt es, vor allem wenn man noch nicht so routiniert ist, zu „merkwürdigen“ Effekten, die einen Programmieranfänger oder

leicht Fortgeschrittenen zur Verzweiflung bringen können; dies um so mehr, wenn man die internen, teils subtilen Zusammenhänge (noch) nicht vollständig versteht.

Fangen wir schrittweise an (siehe auch Abschnitt 30.12):

1. Variablen vom Datentyp `char` belegen meistens ein Byte, d.h. `sizeof(char) == 1`.
2. Alle `char` Konstanten wie beispielsweise `'a'`, `'\t'` und `'\040'` werden vom Compiler in den Typ `int` umgewandelt, auch wenn dies früher anders war.
3. Diese beiden Sachverhalte bedeuten,
   (a) dass beim Vergleich zweier Variablen vom Typ `char` keine Typkonvertierungen durchgeführt werden, und
   (b) dass beim Vergleich mit einer Konstanten die Variable erst auf den Datentyp `int` erweitert wird.
4. Bei allen Vergleichen muss daran gedacht werden, dass aufgrund der internen Repräsentation als 8-Bit Wert, Variablen vom Typ `char` nur Werte von `-128..127` annehmen können. Das bedeutet beispielsweise, dass eine Abfrage `c > 127` im Regelfall nicht besonders sinnvoll ist.
5. Durch die implizite Umwandlung in den Datentyp `int` sind die drei Vergleiche "`c == '\60'`", "`c == 060`" und "`c == 48`" identisch.
6. Bei Wertzuweisung an eine Variable vom Typ `char` sind die Verhältnisse genau umgekehrt: aus dem Wert wird das letzte Byte „ausgeschnitten"[2] und als Ergebnis der Variablen zugewiesen. Beispiele sind:

```
1  c = 60;          // ergibt 60
2  c = -46;         // ergibt -46
3  c = 256;         // ergibt 0 da 256 % 256 = 0
4  c = -256;        // ergibt 0 da -256 % 256 = 0
5  c = 266;         // ergibt 10 da 266 % 256 = 10
6  c = -266;        // ergibt -10 da -266 % 256 = -10
```

30.12 Datentyp `char` auf modernen Architekturen

Im Laufe der Zeit haben sich einige Aspekte bezüglich des Datentyps `char` gewandelt. Insofern trifft das oben gesagte nicht unbedingt in allen Aspekten auf heutige Rechnerarchitekturen zu. Auch wenn es *meistens* so ist, dass die Größe einer `char`-Variablen genau 1 ist, so nehmen sich heutige Compiler die Freiheit, dies zu ändern.

So kann es beispielsweise sein, dass sich ein Compiler dazu entschließt, jede Zeichenvariable in ein Maschinenwort zu packen. Dies könnte dazu führen, dass beispielsweise auf einer 64-

[2] Das Ausschneiden kann man sich am einfachsten mittels `wert % 256` veranschaulichen.

Bit Architektur jede Zeichenvariable acht Bytes belegt, also `sizeof( char )` den Wert `8` liefert.

Momentan ist es so, dass derartige Effekte nur bei Einschalten spezieller Compiler-Optionen auftreten. Der wichtige Punkt ist, dass diese Effekte auftreten können. Um also ganz sicher zu sein, muss dies bei der Verwendung von Arrays (Kapitel 33) auch beim Datentyp `char` berücksichtigt werden.

Kapitel 31

Klassifikation von Zeichen: `ctype.h`

Die ASCII-mäßige Definition sowie Verwendung von Zeichen wurde in den beiden vorherigen Kapiteln besprochen. Nun ist es manchmal sinnvoll, eingelesene Zeichen hinsichtlich ihrer Klassifizierung zu unterscheiden. Dazu zählen beispielsweise Buchstaben, Ziffern und Sonderzeichen. Hierfür sind in der Datei `ctype.h` verschiedene Makros (näheres zu Makros besprechen wir in Kapitel 38.3) wie beispielsweise `isnum(int c)` und `isalpha(int c)` definiert, die das Erledigen dieser Aufgabe recht bequem machen. Da es sich bei der Datei `ctype.h` um eine dieser vorgefertigten Standarddateien handelt, sollte diese mittels `#include <ctype.h>` eingefügt werden.

31.1 Verwendung

In der Datei `ctype.h` der Standardbibliothek sind einige Funktionen und Makros definiert, die das Klassifizieren von Zeichen sehr erleichtern. Da es sich um eine Standardbibliothek handelt, sollte sie mittels `#include <ctype.h>` in das Programm eingefügt werden. Obwohl es sich bei den meisten Definitionen nicht wirklich um echte Funktionen sondern um Makros handelt, können sie dennoch wie Funktionen verwendet werden. Alle Funktionen (eigentlich Makros) geben einen Wert ungleich 0 zurück, wenn das übergebene Zeichen die Bedingung erfüllt; sonst wird der Wert 0 zurückgegeben. Die einzelnen uns zur Verfügung stehenden Makros zur Klassifizierung von Zeichen werden in der Tabelle auf der nächsten Seite zusammengefasst.

Die letzten beiden Funktionen (`tolower()` und `toupper()`) dieser Tabelle sind anders als die anderen. Erstens führen sie keine Tests durch, sondern wandeln Klein- in Großbuchstaben bzw. umgekehrt um. Und zweitens sind diese Funktionen nicht als Makros sondern als richtige Funktionen kodiert. Übrigens, diese beiden Funktionen wandeln nur die betreffenden Buchstaben um, alle anderen Zeichen lassen sie unverändert.

Funktion	Durchgeführter Test auf	Beispiele
`int islower ( int c )`	Kleinbuchstaben	`a-z`
`int isupper ( int c )`	Großbuchstaben	`A-Z`
`int isalpha ( int c )`	Buchstaben	`a-z A-Z`
`int isdigit ( int c )`	Ziffer	`0-9`
`int isxdigit( int c )`	hexadezimale Ziffer	`0-9 a-f A-F`
`int isalnum ( int c )`	alphanumerische Zeichen	`a-z A-Z 0-9`
`int isspace ( int c )`	Leerzeichen/Zwischenraum	`' ' \t \n`
`int ispunct ( int c )`	Interpunktionszeichen	`. , ; + - = {}( ) [ ] < > ~`
`int isgraph ( int c )`	druckbares Zeichen ohne Leerzeichen	ASCII-Tabelle: Zeichen 33-126
`int isprint ( int c )`	druckbares Zeichen mit Leerzeichen	ASCII-Tabelle: Zeichen 32-126
`int iscntrl ( int c )`	Steuerzeichen	ASCII-Tabelle: Zeichen 0-31
`int tolower ( int c )`	Wandelt Groß- in Kleinbuchstaben um	`tolower( 'A' )`
`int toupper ( int c )`	Wandelt Klein- in Großbuchstaben um	`toupper( 'a' )`

In diesem Abschnitt haben wir des Öfteren energisch darauf hingewiesen, dass es sich bei den Definitionen der Standardbibliothek `ctype.h` nicht um Funktionen sondern um Makros handelt. Die Unterschiede sind wichtig, doch deren Diskussion ist für den Programmieranfänger noch zu schwierig, sodass wir diese auf Kapitel 43.8 verschieben.

31.2 Programmbeispiele

Das folgende Programmbeispiel haben wir schon in Kapitel 30 besprochen. Durch die Verwendung des Makros `isalpha()` wird die Abfrage in der `if`-Anweisung leicht vereinfacht.

```
 1 #include <stdio.h>
 2 #include <ctype.h>
 3
 4 int main()
 5     {
 6         char c;
 7         printf( "Bitte ein Zeichen eingeben: " );
 8         scanf( "%c", & c );
 9         if ( isalpha( c ) )
10            printf( "'%c' ist ein Buchstabe\n", c );
11         else printf( "'%c' ist kein Buchstabe\n", c );
12     }
```

In seinem Kern zeigt das folgende Programmstück, wie solange ein Zeichen eingelesen wird

(Schleife von Zeile 11 bis 16), bis es sich tatsächlich um einen Buchstaben handelt. Im Fehlerfalle (Zeilen 14 und 15) wird eine kleine Fehlermeldung ausgegeben. Der Programmcode davor und dahinter ist nicht wirklich sinnvoll und dient nur zur Illustration.

```
 1 #include <stdio.h>
 2 #include <ctype.h>
 3
 4 int main()
 5     {
 6         char c;
 7
 8         //      Hier stehen irgendwelche Anweisungen
 9
10         //      Abfrage, womit weiter gemacht werden soll
11         do {
12              printf( "Bitte neuen Buchstaben eingeben: " );
13              scanf( "%c", & c );
14              if ( ! isalpha( c ) )
15                 printf( "'%c' ist kein Buchstabe\n", c );
16            } while( ! isalpha( c ) );
17
18         printf( "Jetzt wird mit '%c' weiter gemacht\n", c );
19
20         //      Hier stehen andere Anweisungen
21
22     }
```

Kapitel 32

Datentyp `double` für „reelle“ Zahlen

Der Datentyp für „reelle“ Zahlen heißt `double`. Natürlich handelt es sich nicht wirklich um reelle Zahlen, denn die Auflösung auf einem Digitalrechner ist begrenzt. Aber Zahlen vom Typ `double` sind so genau, dass sie für die meisten Anwendungen ausreichen. Innerhalb eines C-Programms können reelle Zahlen in drei verschiedenen Varianten eingegeben werden: ganzzahlige Konstante[1] `1234`, in Fixpunkt-Darstellung `1234.5678` und als Fließkommazahl `-12.34E-3`.

32.1 Verwendung

C-Syntax

```
double d, U;
d = 2.8;
U = d * 3.14;
```

Abstrakte Programmierung

```
Variablen: Typ: Double: d, U
setze d = 2,8
setze U = d * 3,14
```

Hinweise: Bei Verwendung von `double`-Variablen und `double`-Zahlen (Konstanten) sollte folgendes beachtet werden:

1. Bevor eine Variable benutzt wird, muss sie definiert[2] werden. Dies passiert in obigem Beispiel in der ersten Zeile.
2. Da Programmiersprachen in der Regel vom angelsächsischen geprägt sind, wird kein Komma sondern ein (Dezimal-) Punkt verwendet.
3. Eine definierte Variable kann immer nur innerhalb der geschweiften Klammern verwendet werden, in denen sie auch definiert wurde.

[1] Um's mal etwas genauer zu nehmen: Eine ganzzahlige Konstante ist eine Konstante vom Typ `int`. Aber beim Zuweisen an eine reelle Variable merkt dies der Compiler und wandelt die Zahl von sich aus um.

[2] Es geht auch noch ein klein wenig anders, aber das besprechen wir erst in Kapitel 55.

32.2 `double`-Konstanten und interne Repräsentation

Wie eingangs erwähnt, können `double`-Zahlen in drei verschiedenen Formaten angegeben werden. Das erste Format, eine ganze Zahl, haben wir bereits eingehend in Kapitel 21.2 besprochen, sodass wir hier nicht mehr darauf eingehen.

Das zweite Format ist als Fixpunktdarstellung bekannt. Ein paar Beispiele sind: `1.234`, `-1.234`, `+1.234` und `-34.123456`.

Das dritte Format sind die sogenannten Gleitpunktdarstellungen. In diesem Fall wird eine beliebige Fixpunktzahl genommen und ihr mittels `E` oder `e` ein Exponent zur Basis zehn nachgestellt. Alle folgenden Zahlen ergeben den Wert $\pi = 3.141$: `3.141`, `0.3141E1`, `0.3141e1` und `3141E-3`.

Aber egal welches Format man auch immer wählt, im Arbeitsspeicher kommt es immer zur gleichen Bit-Repräsentation. Mit anderen Worten: Die Designer der Programmiersprache C bieten uns drei verschiedene Formate an, damit wir es je nach Problemstellung möglichst einfach haben.

Ferner ist es wie bereits im Kapitel über die ganzen Zahlen gesagt: Egal, wie die Zahlen eingegeben werden, intern (d.h. im Arbeitsspeicher) haben sie alle das gleiche Format. Gemäß des IEEE-Standards [10] wird der Dezimalpunkt so verschoben, dass vor ihm genau eine Ziffer ungleich null ist, und der Wert wird mittels des richtigen Exponenten wiederhergestellt. Durch diese Darstellung haben alle Zahlen vom Typ `double` die selbe Genauigkeit. Wer mehr dazu wissen möchte, schaue bitte in der Literatur nach [10, 13].

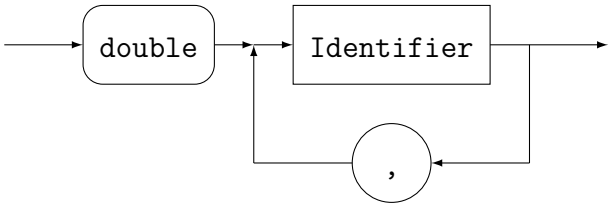
Und hier gilt ebenso wie bereits in Kapitel 21.2: die Zahl der belegten Bytes bekommt man durch die Funktion `sizeof()` heraus, was bei mir auf dem Bildschirm immer acht ist:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5         double d;
 6         printf( "Zahl der Bytes: %d\n", sizeof( double ) );
 7         printf( "Zahl der Bytes: %d\n", sizeof( d ) );
 8         printf( "Zahl der Bytes: %d\n", sizeof( 0.0314E2 ) );
 9         printf( "Zahl der Bytes: %d\n", sizeof( d - 1.0 ) );
10     }
```

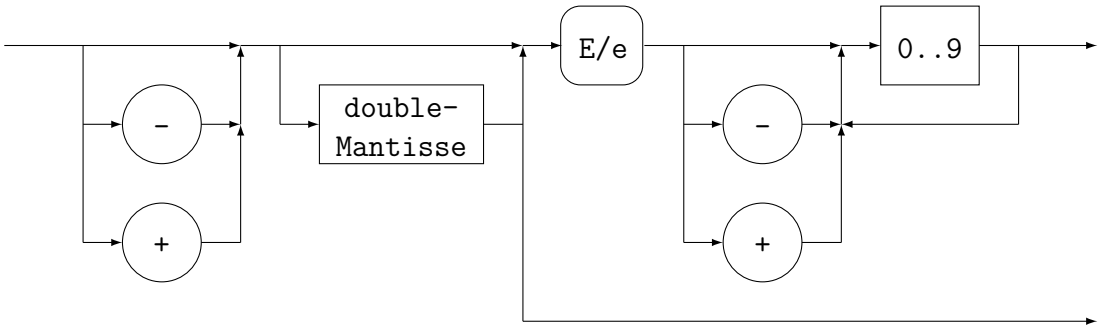
32.3 Syntaxdiagramme (vereinfacht)

Das noch etwas vereinfachte Syntaxdiagramm sieht wie folgt aus:

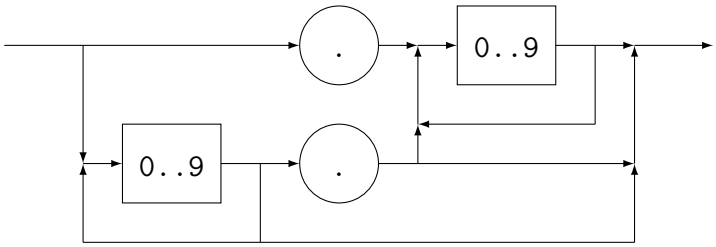
`double-Definition (vereinfacht)`



double-Konstante



double-Mantisse



Für die einzelnen Ziffern-Blöcke wie beispielsweise `0..9` etc. fügen wir hier keine Syntaxdiagramme ein, da deren Bedeutung sehr offensichtlich sein sollte ;-) !

32.4 Korrekte und fehlerhafte Beispiele

Korrekte Beispiele:

```
1 double d, x;
2 d = 1234.E-13;
3 x = 1E-1;
```

Fehlerhafte Beispiele:

```
1 d = 1.fea;  // hex nicht erlaubt
2 d = 1E01E1; // zweimal Exponent
3 d = 1/0;    // nicht sinnvoll
```

32.5 Ausgabe von `double`

Bei der Ausgabe einer Zahl vom Typ `double` kann man zwischen Fixpunkt- und Exponentialdarstellung wählen. Für erstere gibt es die Formatierungsangabe `%a.bf`. Hierbei gibt `a` die Gesamtzahl der zu druckenden Zeichen und `b` die Zahl der Nachkommastellen an. Zum Beispiel führt `printf( "%6.3f\n", -47.1234 )` zur Ausgabe von `-47.123`. Für das zweite Format gibt es die Formatierungsangabe `%a.be`. Hierbei gibt `a` wiederum die Geamtzahl

der zu druckenden Zeichen an und `b` die Zahl der Nachkommastellen. Beispielsweise führt `printf( "%10.3e\n", -47.1234 )` zur Ausgabe `-4.712e+01`.

32.6 Einlesen von `double`-Werten

Auch `double`-Zahlen werden mit der Funktion `scanf()` eingelesen. Allerdings lautet die Formatierung (etwas inkonsistent) `%lf` und *nicht* `%f`. Mittels `scanf( "%lf", & d )` liest man beispielsweise einen Wert für die Variable `d` ein, die vom Typ `double` sein muss.

32.7 Definition einschließlich Initialisierung

Hier gilt das gleiche wie bereits in Kapitel 21.8 gesagt: Variablen vom Typ `double` können bereits bei ihrer Definition initialisiert werden. Beispielsweise `double pi = 3.14;`

32.8 Rechenoperationen und Rundungsfehler

Wie bei Zahlen vom Typ `int` können auch bei `double` die vier Grundrechenarten verwendet werden. Bei `double`-Zahlen gibt es aber ein gravierendes Problem: Da die Auflösung begrenzt ist, gibt es Rundungsfehler! Dies macht Abfragen auf Gleichheit bzw. Ungleichheit immer etwas problematisch, denn sie sind *häufig* immer wahr oder unwahr.

Beispiel: Wir wollen eine `double`-Variable `x` von 0.0 bis 1.0 in zehn Schritten (Schrittweite 0.1) laufen lassen. Da wir unser Programm möglichst allgemein halten wollen, rechnen wir die Schrittweite `dx = (x_ende - x)/10.0` aus. Man könnte auf folgende Lösung kommen:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char ** argv )
 4     {
 5         double x = 0.0, x_ende = 1.0, dx = (x_ende - x)/10.0;
 6         for( ; x != x_ende; x = x + dx )
 7         {
 8             printf( "x= %5.2f\n", x );
 9             if( x > x_ende )  // problem! wir sind viel zu weit!
10                 break;
11         }
12     }
```

„Normalerweise“ müsste die `for`-Schleife (Zeile 6) bei `x == 1.0` abbrechen, genau nach dem zehnten Mal. Doch leider gibt es diese Rundungsfehler und wir erleben folgende Ausgabe: `x= 0.0, ..., x= 1.0, x= 1.1`. Hätten wir nicht die Abfrage in Zeile 9 nebst der `break`-Anweisung eingebaut, hätten wir eine Endlosschleife, die nie aufhört.

Maßnahmen „gegen“ Rundungsfehler: In solchen Fällen ist das einfachste, wenn man die Schleifenbedingung auf ein `>=` bzw. `<=` ändert und eine kleine Toleranz hinzufügt. In unserem Fall ist dies einfach `0.5*dx` in Zeile 6.

```
1  #include <stdio.h>
2
3  int main( int argc, char ** argv )
4      {
5         double x = 0.0, x_ende = 1.0, dx = (x_ende - x)/10.0;
6         for( ; x <= x_ende + 0.5*dx; x = x + dx )
7         {
8            printf( "x= %5.2f\n", x );
9            if( x > x_ende )  // problem! wir sind viel zu weit!
10              break;
11        }
12     }
```

Eine alternative Maßnahme besteht darin, dass wir den aktuellen `x`-Wert jedesmal ausrechnen. In diesem Beispiel gibt es natürlich auch vereinzelte Rundungsfehler, aber diese wirken sich weder auf die Schleife aus noch pflanzen sie sich fort.

```
1  #include <stdio.h>
2
3  int main( int argc, char ** argv )
4      {
5          int i;
6          double x, x_anfang = 0.0, x_ende = 1.0, dx;
7          dx = (x_ende - x_anfang)/10.0;
8          for( i = 0; i <= 10; i++ )
9          {
10            x = x_anfang + i * dx;
11            printf( "x= %5.2f\n", x );
12         }
13     }
```

Kapitel 33

Arrays: Eine erste Einführung

Aus der Mathematik sind wohl jedem Vektoren $\vec{x}$ und Koeffizienten $a_0, a_1, \ldots, a_n$ bekannt. In der Programmierung nennt man solche Dinge ein eindimensionales Array bzw. eindimensionales Feld. In den meisten Programmiersprachen gibt es auch zwei- und mehrdimensionale Arrays; zweidimensionale Arrays bezeichnet man in der Mathematik oft auch als Matrizen. Ein wesentliches Charakteristikum von Arrays ist, dass alle Elemente den selben Datentyp besitzen.

In diesem Kapitel erfolgt nur eine erste kurze Einführung, damit Ihr damit umgehen und erste einfache Programme erstellen könnt. Da es aber noch viele wichtige Details, insbesondere im Zusammenhang mit der internen Darstellung und dem Konzept der Zeiger gibt, werden wir dieses Thema erneut in Kapitel 49 aufgreifen.

33.1 Verwendung

Dieser Abschnitt führt zunächst eindimensionale Arrays phänomenologisch ein.

C-Syntax	**Abstrakte Programmierung**
`char   c[  4 ];`	`Variablen: Typ Array 0.. 3 of Zeichen : c`
`double d[ 12 ];`	`Variablen: Typ Array 0..11 of Double  : d`
`int    a[ 10 ];`	`Variablen: Typ Array 0.. 9 of Ganzzahl: a`
`a[ 0 ] = 3;`	`setze` a_0 `= 3`
`a[ 9 ] = 17;`	`setze` a_9 `= 17`

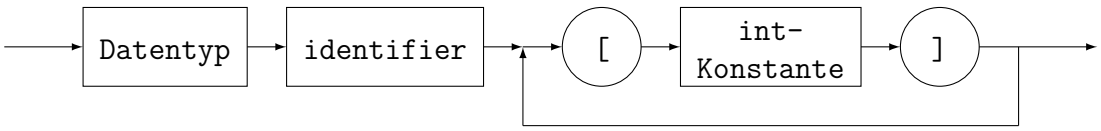
Hinweise: Bei der Verwendung von `arrays` sollte folgendes unbedingt beachtet werden:

1. Bei der Definition eindimensionaler Arrays wird genau eine Größe angegeben, die die Zahl der Elemente definiert.
2. Bei einer Größe N sind die erlaubten Indizies $0, 1, \ldots, N-1$.

3. Vom Compiler wird garantiert, dass alle Elemente bündig aneinander aufgereiht im Arbeitsspeicher abgelegt werden.
4. Arrays können einander *nicht* zugewiesen werden; `int a[ 10 ], b[ 10 ]; a = b;` funktioniert *nicht!* Die Zuweisung muss mittels einer Schleife elementweise erfolgen.
5. Indizes außerhalb des erlaubten Bereichs gehören nicht zum Array. Diese sind funktional ein Fehler, werden vom Compiler aber weder erkannt noch als solche gemeldet.
6. Auch zur Laufzeit werden Zugriffe außerhalb des Arrays nicht überprüft. Diese Zugriffe führen häufig zu sehr unerwünschten Effekten (da sich dort eventuell eine andere Variable befindet), manchmal auch zu einem Programmabsturz. Das Finden derartiger Fehler ist in der Regel zeit- und nervenaufreibend.

33.2 Syntaxdiagramme

array-Definition



Das Syntaxdiagramm zeigt auch, wie man mehrdimensionale Arrays erhält: Für jede weitere Dimension benötigt man ein weiteres Paar eckiger Klammern `[]` nebenst einer eingeschlossenen Größenangabe (Beispiele siehe unten).

33.3 Korrekte Beispiele

```
1 // Definitionen
2
3 int a[ 10 ];
4 char c[ 3 ];
5 double d[ 12 ];
```

```
1 // Zugriff
2
3 a[ 0 ] = 1234;
4 c[ 2 ] = 'a';
5 d[ 1 ] = 12.34;
```

33.4 Fehlerhafte Beispiele

```
1 int a{ 3 };      // falsche Klammern; muessen eckige [] sein
2 a[ -1 ] = 1;     // falscher Index; erlaubt 0..N-1
3 a[ 1.1 ] = 1;    // falscher Index; muss int sein
```

33.5 Ausgabe eines Arrays

Ein Array kann nicht einfach so als Ganzes ausgegeben werden. Dies muss immer elementweise, beispielsweise in einer Schleife, geschehen. Unter bestimmten Umständen sind Arrays vom Datentyp `char` eine Ausnahme. Aber dies behandeln wir erst in Kapitel 50.

33.6 Array-Größen und Änderungsfreundlichkeit

Es sollte ziemlich offensichtlich sein, dass man die Größe eines Arrays an verschiedenen Stellen seines Programms benötigt. Ein erstes offensichtliches Beispiel ist dessen Initialisierung, wie in folgendem Beispiel gezeigt:

```
1  int main( int argc, char **argv )
2      {
3          int i, a[ 10 ];
4          for( i = 0; i < 10; i = i + 1 )
5              a[ i ] = 0;
6      }
```

Mit Sicherheit ist es so, dass derartige Schleifen und Abfragen auf `i < 10` noch öfter im Programm auftauchen. Wie ist das nun mit der Änderbarkeit einer solchen Konstruktion? Antwort: Eher mittelmäßig. Man müsste im Programm jeweils die `10` durch eine andere Zahl ersetzen. Dabei besteht die Gefahr, dass man eine `10` vergisst oder eine andere `10`, die mit diesem Array gar nichts zu tun hat, versehentlich ändert. Daher ist es *ratsam*, die Größe mittels eines separaten `#define` wie folgt zu spezifizieren:

```
1  #define SIZE_A  10
2
3  int main( int argc, char **argv )
4      {
5          int i, a[ SIZE_A ];
6          for( i = 0; i < SIZE_A; i = i + 1 )
7              a[ i ] = 0;
8      }
```

Die Funktionalität sollte intuitiv sein. In der ersten Zeile wird ein „Label“ `SIZE_A` definiert, dessen Auftauchen im weiteren Programm durch seine rechte Seite (bei uns die Zahl `10`) ersetzt wird. Dies wird vom Präprozessor abgewickelt, der der erste Schritt des Compilers ist und den wir genauer in Kapitel 38 erklären. Bei Änderungswünschen braucht man nur die Definition in der ersten Zeile ändern und schon funktioniert nach einem erneuten Übersetzen wieder alles wie gehabt.

33.7 Einlesen von Array-Elementen

Bereits in unserem ersten Beispielprogramm (Kaptiel 7) haben wir kurz erläutert, dass die Eingabefunktion `scanf()` wissen muss, wo sich eine Variable im Arbeitsspeicher befindet und dass wir deshalb die Variable mit einem `&`-Zeichen versehen. Ein Beispiel aus unserem ersten Programm war `scanf( "%d", & a )`. Unter Einbeziehung dessen, was wir in den vorherigen Abschnitten gelernt haben, könnte das Einlesen der Array-Elemente wie folgt aussehen:

```
 1 #include <stdio.h>
 2
 3 #define SIZE_A  10
 4
 5 int main( int argc, char **argv )
 6     {
 7        int i, a[ SIZE_A ];
 8        for( i = 0; i < SIZE_A; i = i + 1 )
 9           scanf( "%d", & a[ i ] );
10     }
```

Mehr zum Thema Adressberechnung und Arrays kommt in Kapitel 49.

33.8 Definition einschließlich Initialisierung

In Kapitel 21 haben wir bereits gezeigt, dass sich Variablen schon direkt bei ihrer Definition initialisieren lassen. Beispielsweise wird durch die Anweisung `int i = 3;` die Variable `i` definiert und unmittelbar mit dem Wert `3` initialisiert. Dieses Konzept kann auch direkt auf Arrays angewendet werden. Nur müssen hier die Initialwerte in geschweifte Klammern gesetzt werden:

```
1 int  a[ 4 ] = { 1, 2, 3, 4 };
2 char c[ 3 ] = { 'a', 'b', 'c' };
```

Die Ausführung ist so, wie man sie sich intuitiv vorstellt. Beispielsweise haben `a[ 0 ]` und `a[ 1 ]` die Werte `1` und `2`.

33.9 Größenfestlegung durch Initialisierung

Man kann die Größe eines Arrays auch durch seine Initialisierung festlegen:

```
1 int a[] = { -1, -2, -3, -4, -5 };
```

Die Funktionsweise ist einfach und naheliegend: Der Compiler „sieht“, dass die Initialisierung aus fünf Konstanten besteht. Entsprechend „weiß“ er, dass das Array genau fünf Elemente benötigt und behandelt diese Definition als stünde folgendes da:

```
1 int a[ 5 ] = { -1, -2, -3, -4, -5 };
```

Nur sieht man als Programmierer diese Größenangabe nicht. Für die Größe könnte man dennoch ein `#define` verwenden, wie wir es oben getan haben. Aber das könnte unbequem werden, denn was ist, wenn dieses `#define` nicht mit der tatsächlichen Größe des Arrays übereinstimmt. Nun, dann fehlen entweder Zahlen oder einige Elemente sind nicht initialisiert. Beides widerspräche der Idee der impliziten Größenfestlegung. Der Ausweg besteht in folgender Modifikation:

```
1 int a[] = { -1, -2, -3, -4, -5 };
2
3 #define SIZE_A           (sizeof(a)/sizeof(int))
```

In Kapitel 21 haben wir bereits erläutert, dass die Compiler-Funktion `sizeof()` die Größe eines Objektes oder eines Datentyps ermittelt. In unserem Fall liefert `sizeof(a)` auf *meinem* Rechner den Wert 20 und `sizeof(int)` den Wert vier, sodass `SIZE_A` den Wert fünf bekommt.

33.10 Mehrdimensionale Arrays etc.

Wie oben geschrieben, dient dieses Kapitel einer ersten Einführung in die Verwendung von Arrays. Mehrdimensionale Arrays und insbesondere die interne Repräsentation sowie Adressberechnungen kommen alle erst in Kapitel 49.

Kapitel 34

Qualitätskriterien

Programmieren ist eine äußerst kreative Tätigkeit, denn ein Programm kann man nicht ausrechnen wie eine einfache Gleichung. Nein, es gibt viele richtige Lösungen und noch viel mehr falsche. Beim Programmieren muss man einen Weg durch das Dickicht finden und ständig Entscheidungen fällen: welche Datenstruktur, welche Reihenfolge der Anweisungen, welche Funktionen usw. und so fort. Anfänglich stehen immer wieder folgende Fragen im Raum: *„Ist meine Entscheidung richtig? Welche Variante muss ich auswählen?“* Sehr schnell jedoch (sofern man die Übungen richtig macht) ändern sich diese Fragen in die folgende Richtung: *„Welche der Varianten ist denn die bessere? Womit kann ich besser punkten?“* Bei den Lernenden, sicherlich auch bedingt durch die langen Antwortzeiten einiger Softwareprodukte, konzentrieren sich die Überlegungen sehr auf die *vermutete* Ausführungsgeschwindigkeit. Vor diesem Hintergrund versucht dieses Kapitel durch die Diskussion einiger wichtiger Qualitätskriterien etwas Orientierung zu geben.

34.1 Korrektheit

Dies war, ist und wird das wichtigste Qualitätskriterium für Software sein. Was nutzt ein Programm, das schnell arbeitet, aber das falsche Ergebnis liefert? Was nutzt ein Airbagsystem, das sich erst nach einem Crash aktiviert, wenn beispielsweise der Krankenwagen eintrifft. Oder welchen Zweck hat ein Lohnbuchhaltungsprogramm, wenn der ausgezahlte Lohn immer viel zu gering ist?

Die Korrektheit eines Programms ist das wichtigste Ziel. Zu diesem Zweck wird extra das Pflichtenheft erstellt, wie wir bereits in Kapitel 6 erläutert haben. Es ist sogar so, dass sich die Korrektheit hiervon ableiten lässt. Wenn in einem derartigen Pflichtenheft sogar stehen sollte, dass eins plus eins drei ergibt, dann *muss* sich das Programm auch so verhalten. Daher ist die Erstellung und Überprüfung des Pflichtenheftes eine bedeutende Aufgabe.

Um es nochmals anders zu formulieren: alle anderen Qualitätskriterien sind zweitrangig! Bei wichtigen kommerziellen bzw. militärischen Programmen wird die Korrektheit nicht

nur anhand einiger Tests empirisch ermittelt sondern mittels strenger mathematischer Methoden bewiesen. Aber diesen Aufwand wollen wir mal nicht betreiben sondern lieber den Informatikern überlassen.

34.2 Änderbarkeit und Wartbarkeit

Diese beiden Kriterien sind für sich genommen zweitrangig, aber in der Praxis eine wichtige Voraussetzung für das oberste Qualitätsziel. Sie bedeuten folgendes:

Wartbarkeit: Oftmals zieht sich die Entwicklung eines Programms oder eines Teils davon über längere Zeit hin. Über die Zeit konvergiert es zu einer schönen und im Auge des Betrachters eleganten Lösung. Trotz aller Freude ist nun aber wichtig, dass man auch nach längerer Zeit noch versteht, was man damals programmiert hat. Hierfür ist es unabdingbar, dass man eine konsistente Einrückstrategie hat, die Namen plausibel und aussagekräftig wählt und alle Anweisungen und Ausdrücke nachvollziehbar formuliert. Gerade die letzteren Aspekte ändern sich mit der zunehmenden Programmiererfahrung. Aber insbesondere am Anfang seiner Programmierkarriere sollte man lieber zu ausführlich als zu knapp formulieren und *kommentieren*. Erst mit wachsender Erfahrung sollte man die kurzen Formulierungen wählen und sich dabei zusätzlich an allgemeine bzw. Firmenstandards halten, damit später auch andere Personen eine Chance haben, die eigenen Programme zu verstehen, Fehler zu finden, diese zu beseitigen und anfallende Anpassungen vorzunehmen.

Beispiel: Die Inhalte zweier Variablen `a` und `b` kann man auf die beiden folgenden Arten tauschen. Welche Variante ist klarer verständlich (links mit Hilfsvariable `h`, rechts ohne)?

```
1 int h; h = a; a = b; b = h;    2 a = a+b; b = a-b; a = a-b;
```

Änderbarkeit: Beide Kriterien, Änderbarkeit und Wartbarkeit, sind stark miteinander verbunden. Aber im Unterschied zu Wartbarkeit bezieht sich Änderbarkeit auf die Möglichkeit, die Funktionalität des Programms zu ändern oder an neue Anforderungen anzupassen.

„Wait a minute! Haben wir nicht eben gesagt, dass sich die Korrektheit anhand des Pflichtenhefts ermitteln lässt? Warum sollen wir anschließend noch etwas ändern? Korrekt ist schließlich korrekt!“ Stimmt natürlich. Aber (!), die Anforderungen an ein Programm können sich ändern. Es können neue Anforderungen hinzu kommen. Und welcher Programmierer wird bei seinem Chef die besseren Karten haben: der, der bei der kleinsten Änderung von vorne mit dem Programmieren anfängt, oder derjenige, der die Anpassungen mittels einiger kleiner Änderungen schnell hin bekommt? Die Antwort ist klar, oder? Um das Ziel einer guten Änderbarkeit zu erreichen, sollten das Programm in möglichst eigenständige Einheiten aufgeteilt werden, die jede für sich eine spezifische Funktionalität unabhängig von allen anderen Komponenten realisiert. Das „Hauptprogramm“ besteht dann nur noch aus der richtigen Kombination dieser Einzelfunktionalitäten.

34.3 Effizienz: Laufzeit und Speicherbedarf

Dies sind natürlich wichtige Dinge, aber weit unwichtiger als man gemeinhin denkt. In Anlehnung an [3] gilt: Wenn ein Programm nicht richtig funktionieren muss, kann man es auch so schreiben, dass es innerhalb einer Millisekunde mit allem fertig ist. Oder in Anlehnung an [3]: Optimieren, lass es bleiben, es sei denn, Du weisst genau, was du tust!

Ganz im Ernst, die Optimierung eines Programms sollte unterschwellig passieren. Eine klare Programmstruktur hilft hier häufig von ganz alleine. Eine Aufteilung in einzelne, möglichst unabhängige Funktionalitäten (siehe oben) führt in der Regel zu weniger unnötigen Aktionen, die von ganz alleine zur Effizienz beitragen.

Desweiteren sollte man sich bei dem Thema Effizienz darüber im klaren sein, dass bei der Beurteilung selbiger häufig von der Länge des Programms bzw. der entsprechenden Teile ausgegangen wird. Dies ist aber grundsätzlich falsch, denn die Annahme ist: ein kurzes C-Programm bedeutet kurzer und schneller Maschinencode. Kann sein, muss aber nicht sein und ist oft auch gar nicht der Fall.

Will man wirklich die Effizienz seines Programms beurteilen und ggf. verbessern, muss man erst einmal schauen, wo denn überhaupt die Ressourcen (Rechenzeit und Arbeitsspeicher) verbraucht werden. Häufig werden Programmteile um 50% optimiert, die nur einmal im Monat für eine Sekunde ausgeführt werden. Dafür werden andere Teile, die ständig Rechenzeit verbrauchen, einfach links liegen gelassen. Das heißt, zuerst ein Profile erstellen, dann die entscheidenden Programmteile identifizieren und schauen, ob sich dort etwas machen lässt. Und beim Optimieren sollte man sich immer im klaren sein, wie der Compiler die einzelnen C-Anweisungen in entsprechenden Maschinencode umsetzt. Aber dies benötigt einiges an Programmiererfahrung, die Ihr zum jetzigen Zeitpunkt noch nicht so habt ...

34.4 Zusammenfassung

Lange Rede, kurzer Sinn, versucht Euch, an folgender *Richtschnur* zu orientieren:

1. Korrektheit zuerst, alles andere kann warten!
2. Klarer und sauberer Entwurf, den sowohl andere als auch Ihr auch nach Jahren noch nachvollziehen könnt.
3. Dokumentation der wichtigsten Entscheidungen.
4. Klares und übersichtliches Ausformulieren mit konsistentem Einrücken und aussagekräftigen Bezeichnern (Variablen- und Funktionsnamen).
5. Optimieren, erst später. Und wenn doch, dann erst nach einem Profiling und einer gründlichen Analyse.

Dann haben auch die Betreuer ihre Freude und alles klappt viel leichter und schneller.

Teil IV

How It Works

Kapitel 35

Ein paar Vorbemerkungen

Mit den beiden vorherigen Skriptteilen haben wir eine gute Basis geschaffen. Wir können die grundlegenden Sprachkonstrukte einüben und bereits einfache Algorithmen programmieren. So weit so gut, aber so ab und zu müssten auch bei Dir Fragen der folgenden Art aufkommen: *„Funktioniert das wirklich so, wie ich es gerade eingetippt habe? Wie kann die CPU das schaffen? Wie funktioniert das eigentlich genau?*

Würden wir jetzt einfach mit dem C-Stoff weiter machen, würden die Probleme zu- statt abnehmen. Hinzu kommt, dass die folgenden C-Konzepte deutlich schwieriger werden. Häufig führt dies dazu, dass Programmieranfänger das Gelernte nur noch rezitieren, ohne es wirklich verstanden zu haben. Und selbst fortgeschrittene Programmierer hoffen dann einfach nur noch, dass ihre Programme schon irgendwie funktionieren werden.

Um hier von vornherein Abhilfe zu schaffen, besprechen wir in diesem Skriptteil, wie die Hardware die einzelnen Dinge abarbeitet und welche Rolle der Compiler dabei spielt. Dem einen oder anderen mag dies viel zu früh erscheinen, Aber es hilft kolossal! Vor allem wenn wir uns Arrays (Kapitel 33 und 49), Funktionen (Kapitel 44 und 47), Zeiger (Kapitel 45 und 46) und später dynamische Datenstrukturen (Skriptteil VII) genauer anschauen. Ohne die Kenntnis über die internen Vorgänge kann man diese Konstrukte einfach nicht wirklich begreifen und bleibt auf der Ebene des Vermutens und Ratens stehen. Zusätzlich besprechen wir in Kapitel 41 ein paar grundlegende Dinge zur Ein-/Ausgabe und klären dabei, wie es zu manch „eigenartigem“ Phänomen kommt. Es geht wie folgt weiter:

Kapitel	Inhalt
36	Die Arbeitsweise von CPU und RAM
37	Der Compiler als Bindeglied zwischen C-Programm und CPU
38	Der Präprozessor `cpp`
39	Der Compiler und seine Arbeitsschritte
40	Die Speicherorganisation durch den Compiler
41	Grundlegendes zur Ein-/Ausgabe

Kapitel 36

Arbeitsweise von CPU und RAM

Bereits auf Seite 34 haben wir im Rahmen des Programmstarts illustriert, wie der Compiler die Dinge neu sortiert und wie diese dann im Arbeitsspeicher (RAM) abgelegt werden. Ferner haben wir *angedeutet,* dass die CPU irgendwie mit dem Arbeitsspeicher in Verbindung steht. In diesem Kapitel diskutieren wir die folgenden Fragen:

Kapitel	Inhalt
36.2	Wie arbeitet der Arbeitsspeicher (RAM)
36.3	Wie arbeitet die CPU ein Programm ab?
36.4	Wie bekommt eine Variable einen Wert?
36.5	Wie wird ein Ausdruck abgearbeitet?
36.6	Zusammenfassung

Doch zuvor wiederholen wir ein paar wesentliche Sachverhalte aus den vorherigen Kapiteln, die wir wieder einmal als Fragen formulieren.

36.1 Wiederholung: Was kann und weiß die CPU

Für die folgenden Fragen müssen wir uns insbesondere an Kapitel 5 zurückerinnern.

- *„Was weiß die CPU über C und andere Programme?“*
  Nichts!
- *„Was weiß die CPU über mein Programm?“*
  Nichts!
- *„Aber ich bin doch der Besitzer und damit Herrscher über die CPU?“*
  T.W. oder „Träum' weiter“
- *„Die CPU erkennt von sich aus, was im Speicher steht und was das alles soll?“*
  S.W. oder „Schön wär's“! Nein, die CPU arbeitet nur das Programm ab und macht

sich *keine* eigenständigen Gedanken. Die CPU sieht nur, dass im Speicher viele Nullen und Einsen stehen.

– *„Die CPU arbeitet Zeile für Zeile ab?“*
  Gar nicht mal so verkehrt, aber dennoch nicht richtig.
– *„Aber die CPU kann sehr schnell Rechnen sowie Daten hin und her kopieren?“*
  Richtig!
– *„Blöd!“*
  Stimmt, aber dafür rasend schnell!

36.2 Aufbau und Organisation des RAM

Wir haben lange überlegt, wie wir Euch ohne all zu großen Ballast die Organisation des Arbeitsspeichers, auch RAM[1] genannt, erklären können. Wir sind zu folgender bildlichen Erklärung gekommen: Der Arbeitsspeicher besteht aus einer Vielzahl von Bytes mit jeweils acht Bits, die ihrerseits Werte von null und eins annehmen können. Ein Byte ist „zufällig“ die Größe eines Zeichens vom Typ `char`, und mit acht Bits kann man $2^8 = 256$ verschiedene Werte darstellen.

Soweit, so gut. Von der *organisatorischen* Seite her, entspricht der Arbeitsspeicher einem sehr großen Regal, bei dem man immer *genau* eine Schublade öffnen kann. Und nur aus dieser geöffneten Schublade kann man etwas herausnehmen oder hineinlegen; alle anderen Schubladen sind geschlossen und nicht zugänglich.

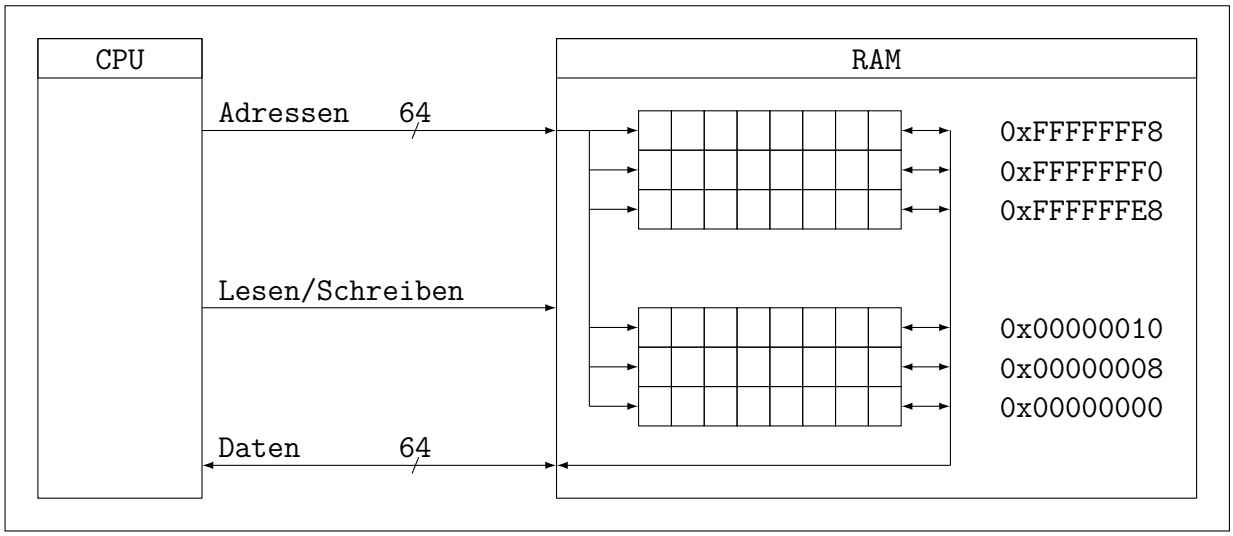
Will man nun „etwas“ von einer Schublade in eine andere packen, so muss man erst eine Schublade öffnen, kann „etwas“ heraus nehmen (einen Zahlenwert), das „etwas“ irgendwo ablegen, die Schublade schließen, eine andere öffnen, das „etwas“ (also die Zahl) von der Ablage holen und in diese Schublade hinein legen, die Schublade wieder schließen und fertig. Klar, oder?

Bevor jetzt einige protestieren: Bei heutigen Speicherbausteinen und modernen 64-Bit Prozessoren können bis zu acht Schubladen *gleichzeitig* geöffnet werden. Aber (!) diese acht Schubladen müssen nebeneinander liegen und müssen zusammen ein gemeinsames etwas (also einen Wert) beherbergen. Im Rahmen unserer Vorlesung abstrahieren wir von diesem für uns unwichtigen Detail und sagen einfach, dass einige Schubladen schmaler, andere etwas breiter sind, aber immer nur eine Schublade geöffnet werden kann.

Die technische Realisierung ist im Bild auf der nächsten Seite wiedergegeben: Zur Kommunikation zwischen Prozessor (CPU) und Arbeitsspeicher gibt es drei Busse: Adress-Bus (oben), Steuer-/Kontroll-Bus (mitte) und Datenbus (unten). Mittels der (in diesem Fall) 64 Adressleitungen wählt die CPU eine ganze Speicherzeile aus. Bildlich gesprochen öffnet

[1] RAM und Arbeitsspeicher werden zwar sehr häufig synonym benutzt, sind aber nicht das gleiche. Die Unterschiede sind hier egal; Ihr werdet sie sicherlich noch im Laufe Eures Studiums erfahren; oder ihr fragt einfach mal nach der Vorlesung.

Organisation des Arbeitsspeichers (RAM)



sich dadurch die entsprechende Schublade, deren Breite über zusätzliche Steuerleitungen definiert wird, die aber im Bild nicht eingezeichnet sind. Über den Steuerbus teilt die CPU dem Arbeitsspeicher mit, ob sie etwas schreiben (in die Schublade legen) oder lesen (etwas aus der Schublade heraus nehmen) will. Die eigentlichen Daten werden über den Datenbus transportiert; in der CPU befinden sich diese Daten in der oben erwähnten Zwischenablage.

Mit 64 Adressleitungen kann man die Kleinigkeit von 18'446'744'073'709'551'616 Bytes eindeutig adressieren. Dies sind etwa vier Milliarden 4 GB RAM Module. Selbst wenn ein derartiges RAM Modul nur einen Euro kosten würde, wäre der Gesamtspeicher für die meisten von uns dennoch zu teuer ...

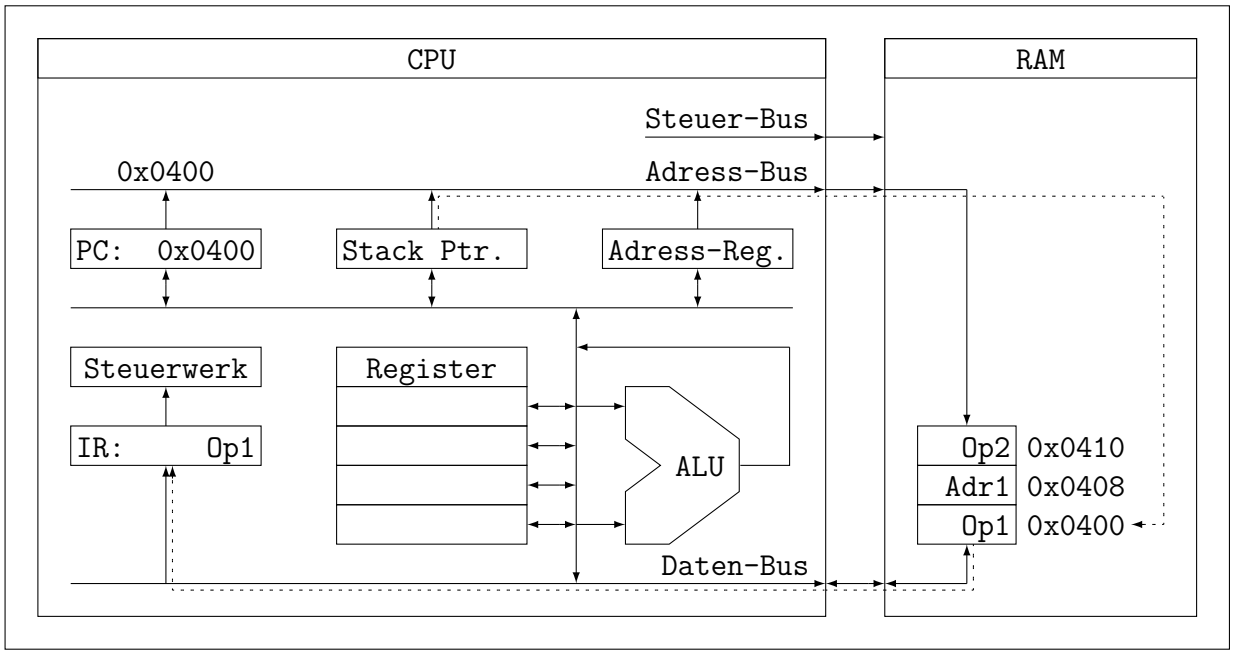
36.3 Wie arbeitet die CPU ein Programm ab?

Im letzten Abschnitt haben wir besprochen, dass man auf den Arbeitsspeicher nur Zelle für Zelle zugreifen kann und nicht alle auf einmal sieht. Also kann auch die CPU ein Programm nur Schritt für Schritt sehen und abarbeiten.

„Aber was heisst hier Schritt für Schritt? Heißt das Zeile für Zeile? Das stünde doch im Widerspruch zu dem, was wir bisher gelernt haben: Gemäß Kapitel 19 sind Leerzeichen und Zeilenwechsel nur dazu da, dass wir durchblicken. Ausserdem weiß der Prozessor nichts über C und kann Programme aller Programmiersprachen abarbeiten ...“ Sehr richtig bemerkt!

Wie schon des Öfteren erwähnt, wandelt der Compiler alle C-Anweisungen in entsprechende Maschineninstruktionen (Op-Codes, Kapitel 5) um. Meist sind das mehr als ein Op-Code pro C-Anweisung, aber manchmal fasst er auch mehrere zusammen. Mit anderen Worten: Das Maschinenprogramm hat eine andere Struktur als das ursprüngliche C-Programm.

Minimaler Aufbau einer CPU



Und „Schritt für Schritt“ heißt nun Maschineninstruktion für Maschineninstruktion.[2]

Um nun die Frage beantworten zu können, wie das Abarbeiten der einzelnen Instruktionen geht, haben wir wieder 'mal ein Bildchen gemalt, das, wie oben zu sehen ist, etwas genauer in die CPU schaut. Als erstes wieder einmal: Don't panic. Wir schauen uns alles einzeln an. Vorweg: Der Arbeitsspeicher ist aus Platzgründen sehr klein ausgefallen. Hervorgehoben sind nur drei Speicherstellen mit ihren Inhalten.

Damit die CPU das Programm schrittweise abarbeiten kann, muss es wissen, wo es gerade ist. Dazu hat es den PC (engl. *Program Counter*). In ihm steht immer die Adresse der nächsten Anweisung. Am Anfang jedes Zyklus wird der Inhalt des PCs (in diesem Fall `0x400`) auf den Adress-Bus gelegt, sodass sich im Arbeitsspeicher die entsprechende Schublade öffnet. Da die CPU den Steuerbus auf „Lesen“ stellt, wird der Inhalt aus der Speicherzelle genommen (in diesem Fall `Op1`) und über den Daten-Bus in die CPU und dort in das Befehlsregister (engl. *Instruction Register*) geladen. Von da aus geht es in das Steuerwerk, in dem alle weiteren Aktionen veranlasst werden. Wichtig: Nach jedem Zugriff des PCs auf den Speicher wird sein Inhalt um eine Schublade weitergeschaltet, was bei unseren 64-Bit Rechnern genau acht Bytes sind.

Nun könnte es sein, wie im Arbeitsspeicher angedeutet, dass noch Parameter nachgeladen werden müssen. Diese kommen dann meistens in andere Register, hauptsächlich in die Register der ALU oder in das angedeutete `Adress-Register`.

So, jetzt noch zu den anderen Teilen. Also, wie gesagt, das Steuerwerk hat alle Fäden in der

[2] Das ist sogar auch dann so, wenn man einen Mehrkern-Prozessor hat.

Hand und sorgt dafür, dass alle Komponenten in der richtigen Art und Weise zusammen arbeiten. Hier werden die einzelnen Befehle (Maschineninstruktionen) in die entsprechenden Hardware-Aktionen umgesetzt.

Dann haben wir noch die ALU. Dieses Wort steht für *Arithmetic-Logic Unit* und wird im Deutschen auch als Rechenwerk bezeichnet. Hier finden alle logischen und arithmetischen Operationen statt. Und die vier angedeuteten Register sind die Zwischenablagen, von denen wir oben gesprochen haben.

„Ähm, könnte das jetzt jemand nochmals bildlich zusammenfassen?“ Klar, kein Problem. Das ist wie beim Einchecken am Flughafen. Die ganzen Leute, die einchecken wollen, sind die einzelnen Instruktionen des Maschinenprogramms, die darauf warten, abgefertigt zu werden. Der PC ist die nette Dame hinter dem Schalter, die ab und zu ruft: *„Next, please!“* Daraufhin geht der nächste Fluggast aus der Warteschlange zur Abfertigung. Und das Steuerwerk ist das Back-Office, das die ganze „Schmutzarbeit“ wie Gepäck sortieren, zum richtigen Flugzeug bringen, einladen, Maschine auftanken etc. pp. erledigen muss.

Wenn wir uns nochmals die drei Speicherinhalte aus obigem Bild anschauen, dann würde das ungefähr wie folgt ablaufen (der Pfeil zeigt immer dahin, wo sich gerade etwas ändert):

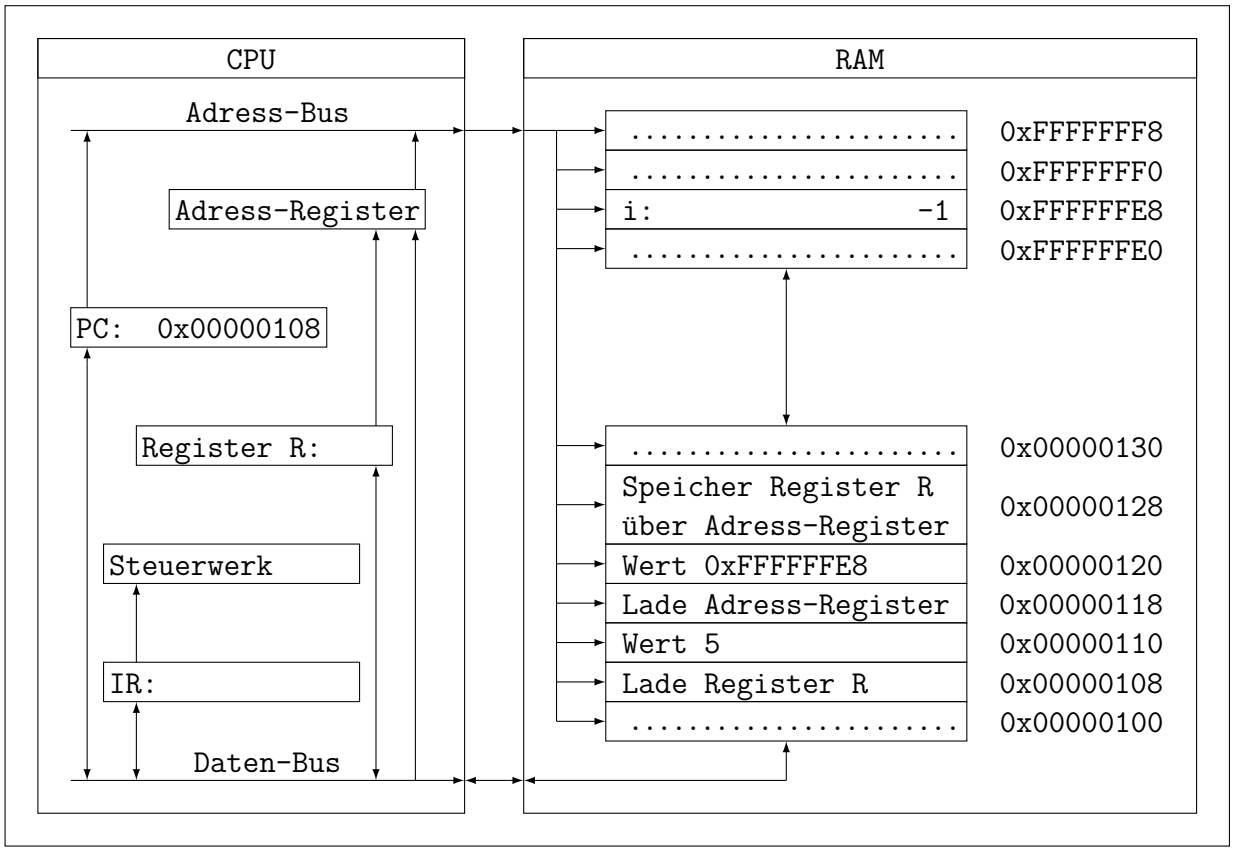
Zeitschritt	PC	IR	Adress-Register
1	→ `0x400`	...	...
2	`0x400`	→ `Op1`	...
3	`0x408`	`Op1`	...
4	`0x408`	`Op1`	→ `Adr1`
5	→ `0x410`	`Op1`	`Adr1`
6	`0x410`	→ `Op2`	`Adr1`

36.4 Wie bekommt eine Variable einen Wert?

Mit dem Wissen der beiden vorherigen Abschnitte ist das kinderleicht, bzw. prozessorleicht. Nehmen wir mal folgende komplexe C-Anweisung: `int i; i = 5;`. Aus dem bisher Gesagten wissen wir, dass es mindestens drei „Zutaten geben muss“: Erstens, irgendwo muss die Konstante `5` sein, irgendwo die Variable `i` und irgendwo die eigentliche Zuweisung. Und da wir wissen, dass immer nur eine Schublade im Arbeitsspeicher aufgehen kann, muss die `5` also zunächst in die Zwischenablage (Register) in der CPU und dann zurück in den Arbeitsspeicher, und zwar in die Schublade, in der sich die Variable `i` befindet.

Genau, das ist alles. Eine Kleinigkeit noch: So eine Maschineninstruktion kann in der Regel nicht beide Datentransporte übernehmen, sodass hierfür zwei Instruktionen benötigt werden. Ein mögliches Programm eines hypothetischen Prozessors könnte so aussehen, wie oben im Bild illustriert ist. Eine gute Übung für jeden Leser wäre es nun, den folgenden zeitlichen Ablauf zu komplettieren, wie wir es eben im vorherigen Abschnitt gemacht haben;

Wertzuweisung an eine Variable



im Bild befindet sich in der Schublade von `i` noch der Wert `-1`:

Das Programm, das sich im Arbeitsspeicher zwischen den Adressen `0x00000108` und `0x00000128` befindet, weist der Variablen `i` den Wert `5` zu. Die beiden `Lade`-Befehle gehen davon aus, dass der nächste Parameter (Wert `5` bzw. die Adresse `0xFFFFFFE8`) in der nächsten Speicherstelle zu finden ist. Der Befehl in der Speicherzelle `0x00000128` ist genauso groß wie alle anderen, nur der Text benötigt mehr Platz. Ferner ist die CPU ein wenig vereinfacht dargestellt.

Im folgenden Ablaufplan sehen wir nochmals, was sich zu welcher Zeit wo ändert. Es sei jedem Leser empfohlen, sich die einzelnen Schritte nochmals klar zu machen. Zur Erinnerung: die Materie ist nicht einfach.

Um hier nicht den falschen Eindruck aufkommen zu lassen: Diese Abschnitte hier dienen dazu, Euch einen Eindruck davon zu vermitteln, wie ein kleines C-Programm durch die Hardware abgearbeitet wird. Bei heutigen Prozessoren heißen die Maschineninstruktionen natürlich anders, aber dennoch ist das Prinzip sehr dicht an der Wirklichkeit und für uns im Rahmen der Lehrveranstaltung präzise genug.

Schritt	PC	IR	Adress Register	Register R	RAM 0xF..E8
1	→ 0x108	...	...	...	-1
2	0x108	→ Lade R. R.	...	...	-1
3	→ 0x110	Lade R. R.	...	...	-1
4	0x110	Lade R. R.	...	→ 5	-1
5	→ 0x118	Lade R. R.	...	5	-1
6	0x118	→ Lade A. R.	...	5	-1
7	→ 0x120	Lade A. R.	...	5	-1
8	0x120	Lade A. R.	→ 0xFFFFFFE8	5	-1
9	→ 0x128	Lade A. R.	0xFFFFFFE8	5	-1
10	0x128	→ Speicher R. R.	0xFFFFFFE8	5	-1
11	→ 0x130	Speicher R. R.	0xFFFFFFE8	5	-1
12	0x130	Speicher R. R.	0xFFFFFFE8	5	→ 5

36.5 Wie wird ein Ausdruck abgearbeitet?

Hier kommt im Wesentlichen nichts Neues mehr hinzu. Die CPU muss nacheinander die Konstanten, die im Ausdruck angegeben sind, aus dem Speicher auslesen (Schublade öffnen und lesen) und in eines der internen Register (Zwischenablage) ablegen. Ferner müssen die Speicherstellen ausgelesen werden, in denen sich die Variablen befinden. Alle Bestandteile müssen am Ende noch richtig „zusammengerechnet“ werden, was die ALU übernimmt. Auch für dieses Zusammenrechnen müssen im Arbeitsspeicher die *richtigen* Instruktionen stehen, sonst kommt ein falsches Ergebnis heraus, aber das sollte klar sein. Zum Schluss muss das Ergebnis noch in die richtige Variable (Speicherzelle) zurückgeschrieben werden. Das war's dann auch schon.

36.6 Zusammenfassung

In diesem Kapitel sollten wir folgende Dinge gelernt haben:

1. Der Arbeitsspeicher besteht aus einer großen Anzahl Speicherstellen (Schubladen), die unterschiedliche Breiten haben können und von der immer nur eine gelesen bzw. beschrieben (geöffnet) werden kann.
2. Die CPU hat zwei zentrale Register. Das sind der Program Counter (PC), der immer auf die Adresse des nächsten Befehls zeigt und das Instruction Register (IR), in dem sich der aktuelle Befehl befindet, der gerade abgearbeitet wird.
3. Auch ein Maschinenprogramm wird Schritt für Schritt abgearbeitet.
4. Die Koordination *aller* Elemente innerhalb der CPU und des Speichers wird vom Steuerwerk übernommen. Dieses Steuerwerk „weiß“ für jede einzelne Maschinenin-

struktion, was alles genau in welcher Reihenfolge gemacht werden muss.

5. Die ALU (Arithemtic-Logic Unit) ist diejenige Komponente, die alle Rechnungen ausführen kann.
6. Die Adressen der Variablen, die wir in unserem C-Programm benutzen, stehen zusammen mit den Instruktionen im Programm (im Maschinencode).
7. Für *jeden* Datentyp (egal ob `int`, `char`, `double` usw.) muss die CPU wissen, wie viele Bytes er groß ist und welche Operationen er in der ALU ausführen muss.

Kapitel 37

Der Compiler als Bindeglied zur CPU

Wie mittlerweile klar geworden sein sollte, übersetzt der Compiler ein Programm, das in einer höheren Programmiersprache wie C geschrieben ist, in die Maschinensprache unseres Zielrechners. Dazu muss er nicht nur die entsprechenden Maschineninstruktionen heraussuchen, sondern auch ein Speicherlayout definieren, damit alle Variablen ihre eigenen Platz bekommen. Um dies alles zu bewerkstelligen legt der Compiler im Laufe des Übersetzens eine Reihe von Tabellen an, mit deren Hilfe er zu oben angedeutetem Layout kommt.

Bevor wir ins Detail gehen, hier nochmals zwei wesentliche Punkte aus der Zusammenfassung des vorherigen Kapitels (leicht umformuliert):

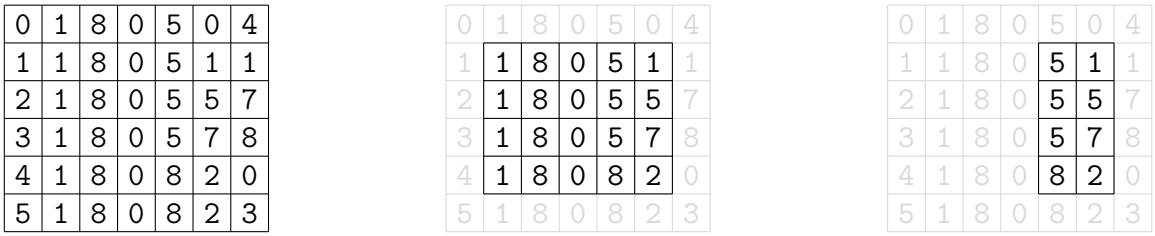
1. Alle Variablen haben einen Ort und damit eine Adresse im Arbeitsspeicher. Diese Adressen müssen für jeden Variablenzugriff in geeigneter Form im Maschinenprogramm stehen.
2. Für jede Variable muss die CPU wissen, wie groß sie ist (wie viel Speicherplatz sie belegt) und von welchem Typ sie ist, damit nachher das Speicherlayout stimmt und immer die richtigen Anweisungen für die ALU ausgewählt werden.

37.1 Motivation: alles nur Nullen und Einsen

Auch wenn den meisten das grundsätzliche Problem bereits klar sein sollte, diskutieren wir hier dennoch nochmals zwei weitere Beispiele, um das Problem so präsent wie möglich zu machen. Als Programmieranfänger will man immer wieder annehmen, dass man doch im Arbeitsspeicher sieht, wo was steht. Aber das ist einfach falsch. Im Arbeitsspeicher stehen nur Nullen und Einsen, die jeweils in Gruppen von acht Bits zu einem Byte zusammen gefasst sind. Ok, das weiß jeder. Und aus dem vorherigen Kapitel ist hoffentlich klar geworden, dass bei modernen Rechnern bis zu acht dieser Bytes (Schubladen) zu einem

Datensatz (einer ganz breiten Schublade) zusammengefasst werden können. Aber was wie zusammengefasst wird und was wo steht kann die CPU nicht wissen.

Beispiel 1: Nehmen wir mal an, wir würden in den Arbeitsspeicher schauen. Dann würden wir viele Zahlen sehen, die wir ins Dezimalsystem konvertiert haben, um uns die Arbeit etwas einfacher zu machen. Wenn wir uns jetzt das linke Teilbild einmal anschauen, dann sehen wir, dass wir nicht viel Sinnvolles sehen. Es könnte sich vielleicht um die Bestellnummern eines großen schwedischen Möbelhauses handeln. Vielleicht aber um etwas anderes, wer weiß das schon.



Schauen wir nun mal auf das mittlere Bild, in dem die inneren Zahlen ein wenig hervor gehoben sind. Dann sehen wir plötzlich vier Zahlen, bei denen es sich um die Postleitzahlen der Universität Rostock, zwei Innenstadtbezirken sowie die von Bentwisch bzw. Gelbensande handeln könnte.

Schauen wir jetzt nach rechts. Dort haben wir andere Zahlen etwas hervorgehoben. Dabei könnte es sich um die internationalen Telefonvorwahlen von Peru, Brasilien, Kolumbien und Korea handeln. Wer weiß. Eigentlich kann es nur der Programmierer bzw. sein C-Programm wissen. Wenn wir aber nur in den Speicher schauen, sehen wir, dass wir viel sehen aber eben nichts erkennen.

Beispiel 2: Nehmen wir mal an, wir würden im Arbeitsspeicher folgende Zahl finden `0x41` und ich würde Euch fragen, was das sein könnte. Einige würden vielleicht sagen: *„Das ist eine Speicheradresse, denn die haben wir immer als Hex-Zahl dargestellt.“* Stimmt. Andere würden vielleicht sagen: *„Nein, das ist doch eine* ***`int`****-Zahl mit dem Wert 65.“* Stimmt auch. Und einer aus der letzten Reihe würde vielleicht behaupten: *„Nein, das ist ein großes* ***`'A'`****, ich habe eben nochmals in der ASCII-Tabelle nachgeschaut.“* Hmmm. Wer hat nun recht?

Es könnten alle Recht haben aber ebenso auch alle falsch liegen. Worum es sich handelt, kann man nicht mehr erkennen, wenn man in den Arbeitsspeicher schaut. Dort befinden sich nur Nullen und Einsen, deren Herkunft und deren Bedeutung man nicht erkennen kann. Es könnte alles sein.

Schlussfolgerung: Datentypen: Die Bedeutung einer konkreten Null-Eins Kombination kann man nur erkennen, wenn man weiß, wie man diese zu interpretieren hat. Mit anderen Worten: Bei jedem Wert im Arbeitsspeicher *muss* man wissen, durch welche Brille man die Nullen und Einsen anschauen muss.

37.2 Grundkonzept: Datentypen

Da das Konzept des *Datentyps* von so grundlegender Bedeutung ist, wiederholen wir hier nochmals die Schlussfolgerung aus dem vorherigen Abschnitt: Die Bedeutung einer konkreten Null-Eins Kombination kann man nur erkennen, wenn man weiß, wie man diese zu interpretieren hat. Mit anderen Worten: Bei jedem Wert im Arbeitsspeicher *muss* man wissen, durch welche Brille man die Nullen und Einsen anschauen muss.

Erst durch diese Interpretationsbrille erschließt sich ein Sinn der entsprechenden Null-Eins Kombination. Und genau diese Brille nennt man *Datentyp.* Ein Datentyp legt die Brille und damit die Bedeutung der einzelnen Nullen und Einsen fest. Und zu jeder Zahl, zu jeder Variablen und zu jedem beliebigen Ausdruck *muss man immer* diese Typinformation parat haben, sonst sieht man nur Zahlensalat und die ALU/CPU kann nicht die richtigen Instruktionen auswählen, um die arithmetischen Operationen auszuführen.

Da dieser Punkt von so grundlegender Bedeutung ist, hier das ganze nochmals etwas anders formuliert. Durch den Datentyp ...

1. ... weiß der Compiler, wie er welchen Wert durch welche Bit-Kombination ablegen bzw. interpretieren soll.
2. ... weiß der Compiler, wie viele Bytes ein Wert hat (wie breit die Schublade ist) und wie viele Bytes jeweils zwischen Arbeitsspeicher und CPU transferiert werden müssen.
3. ... kann der Compiler Adressen im Arbeitsspeicher festlegen. Aus offensichtlichen Gründen dürfen sich die einzelnen Variablen im Arbeitsspeicher nicht überlappen.
4. ... kann der Compiler bei arithmetischen Ausdrücken die richtige Maschineninstruktion für die ALU auswählen.
5. ... merkt der Compiler, ob an einer arithmetischen Operation verschiedene Datentypen beteiligt sind und kann die Repräsentation der Werte entsprechend anpassen.

37.3 Die Compiler-Funktion `sizeof()`

Aus den bisherigen Ausführungen sollte ziemlich klar geworden sein, warum das Konzept *Typ* so wichtig für eine höhere Programmiersprache wie C ist. Intern verwendet der Compiler eine Funktion, mittels derer er den Speicherbedarf jedes einzelnen Typs, jeder Variablen, jedes Wertes und jedes Ausdrucks ermitteln kann.

Hin und wieder müssen wir auch in unserem Quelltext wissen, wie groß ein Typ oder eine Variable ist. So einen Fall hatten wir bereits im Zusammenhang mit Arrays in Kapitel 33. Hierfür stellt uns der Compiler die interne Funktion `sizeof()` zur Verfügung, da er sie ohnehin schon intern verwendet. Da es sich bei `sizeof()` um eine Compiler-Funktion

handelt, wertet er entsprechende Funktionsaufrufe schon während der Übersetzung aus und ersetzt die Funktionsaufrufe durch die ermittelten Ergebnisse.

Wie oben schon anklang, kann man die Funktion `sizeof()` auf Typen, Variablen und Ausdrücke anwenden. Das folgende Programm illustriert diese Fälle und gibt auf meinem Rechner drei Mal die Zahl `4` aus, da es sich in allen drei Fällen um den Typ `int` handelt:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4     {
5         int i;
6         printf( "%d\n", sizeof( int ) );
7         printf( "%d\n", sizeof( i ) );
8         printf( "%d\n", sizeof( i + 2 ) );
9     }
```

37.4 Variablen, Werte, Typen und Operationen aus Sicht des Compilers

Da aus den oben erwähnten Gründen die Typinformation so immanent wichtig ist, treibt der Compiler einen sehr großen Aufwand, um beim Übersetzen diese zu erkennen und zu verwalten. Als illustrierendes Beispiel nehmen wir die beiden folgenden Variablendeklarationen an: `int i; double factor;`. Beim Übersetzen baut der Compiler eine große Tabelle auf, in der er für jede Variable alle möglichen Informationen speichert. Dazu gehören unter anderem der Name der Variable, ihr Typ und ihre Adresse im Arbeitsspeicher:

```
Name   : i                    Name   : factor
Typ    : int                  Typ    : double
Adresse: 0xFFFFFFE0           Adresse: 0xFFFFFFD8
Größe  : 4 Bytes              Größe  : 8 Bytes
```

Und dies macht der Compiler sogar ebenso für alle Konstanten und Funktionen:

```
Name   : 5                    Name       : main
Wert   : 5                    Typ        : function
Typ    : const int            Adresse    : 0x08048474
Adresse: xxxxxxxx             Return Typ: int
Größe  : 4 Bytes              Größe      : xxxxxxxx
                              Parameter  : P-Liste
```

Wie unschwer zu erahnen ist, sind hier die Zahlen `5` und die Funktion `main()` beschrieben. Nur wurden hier die Adresse der Konstanten sowie die Größe der Funktion nicht angegeben, da diese der Compiler erst später bestimmen kann. Und letztlich macht der Compiler dies

auch für die Operationssymbole etc. Mit anderen Worten, eine Anweisung wie `i = j + 3;` wandelt der Compiler ungefähr wie folgt um:

`i = j + 3;` ⇒ $\texttt{i}_{\texttt{int}} =_{\texttt{int}} \texttt{j}_{\texttt{int}} +_{\texttt{int}} 3_{\texttt{int}};$

Aus der folgenden Anweisung mit zwei `double`-Variablen wird folgendes:

`factor = d + 3.14` ⇒ $\texttt{factor}_{\texttt{double}} =_{\texttt{double}} \texttt{d}_{\texttt{double}} +_{\texttt{double}} 3.14_{\texttt{double}}$

Mittels dieser Zusatzinformationen, auch *Annotationen* genannt, kann der Compiler alles so umsetzen, dass es richtig funktioniert.

37.5 Implizite und explizite Typumwandlung (`cast`)

Implizite Typumwandlung: *„Und was ist, wenn ich in einem Ausdruck verschiedene Datentypen verwende? Das darf ich doch gemäß Skript Teil III!“* Richtig, das darfst Du, keine Frage. Mittels der oben besprochenen Annotationen ist dies für den Compiler sogar recht einfach. Er muss nur zuvor alle Operanden in den selben Datentyp wandeln und kann dann den entsprechenden Operator auswählen. Das könnte man wie folgt visualisieren:

`factor = i + 3.14;` ⇒ $\texttt{factor}_{\texttt{double}} =_{\texttt{double}} \texttt{(double)}(\texttt{i}_{\texttt{int}})_{\texttt{double}} +_{\texttt{double}} 3.14_{\texttt{double}};$

Diese vom Compiler durchgeführte Anpassung nennt man auch *implizite Typumwandlung.* Sofern notwendig, wird dies vom Compiler automatisch durchgeführt. Aber er kann dies nur durchführen, wenn er durch die anderen Bestandteile des Ausdrucks genügend Informationen hierfür hat.

Explizite Typumwandlung `cast`: Auch wenn der Compiler die Typumwandlung selbstständig durchführt, kann er dies nicht immer; manchmal muss man ihm dabei mittels der *expliziten Typ-Umwandlung* helfen. Hierfür muss man den gewünschten Typ – in Klammern gesetzt – links vor den zu konvertierenden Ausdruck setzen. Obiges Beispiel hätten wir explizit wie folgt schreiben können:

`factor = i + 3.14;` ⇒ `factor = (double)(i) + 3.14;`

Zur weitere Illustration hier gleich noch ein paar weitere Beispiele:

```
1 int i;                  char c;          double d;
2
3 i = (int)   d;          c = (char) i;    d = (double)       i;
4 i = (int) 3.5;          c = (char) d;    d = (double) (i + 2);
5 i = (int) (2.5 * 3.5);
```

Wie schon mehrfach angedeutet, würde in obigen Beispielen der Compiler die Typumwandlungen immer eigenständig durchführen. Aber unsere Hilfe durch die Angabe einer expliziten Typumwandlung schadet nie.

Abläufe bei der Typumwandlung: Der wesentliche Effekt einer Typumwandlung, egal ob nun implizit oder explizit, ist, dass der Compiler Instruktionen absetzt, die eine Anpassung der Bit-Länge (Speicherplatzbedarf) durchführen. Mit anderen Worten: Durch das Absetzen einiger zusätzlicher Instruktionen werden die Schubladenbreiten angepasst.

Technisch gesehen werden beim Anpassen der Speicherbreite entweder einige Bits vorne werterhaltend ergänzt oder möglichwerweise wertändernd abgeschnitten. Der Erste Fall ist unproblematisch, beim zweiten gehen möglicherweise „Informationen“ verloren. Beispielsweise liefert `(char) 256` den Wert `0`, da ein `char` nur acht Bit hat und bei der Zahl `256` die unteren acht Bits immer null sind.

Ein zweiter Effekt tritt auf, wenn man zwischen `double` und Ganzzahltypen (beispielsweise `int` und `char`) hin- und herwandelt. In diesen Fällen werden vom Compiler zusätzlich Instruktionen abgesetzt, um zusätzlich die Bit-Repräsentation anzupassen. Mit anderen Worten: Hier wird eine vollständige Umwandlung vollzogen, da sich die Bit-Repräsentationen von `double` und Ganzzahltypen drastisch voneinander unterscheiden. Bei der Typumwandlung von `double` nach `int` und `char` werden zusätzlich alle Nachkommastellen radikal abgeschnitten.

37.6 Implizite Typumwandlung und Funktionen

Im letzten Abschnitt haben wir gelernt, dass der Compiler bei Ausdrücken eine implizite Typumwandlung bzw. -anpassung vornimmt. Nun könnte ein gewitzter Student auf folgende Frage kommen: *„Warum soll ich mich denn darum kümmern, wenn's eh immer automatisch passiert?“* Unsere Antwort lautet: *„Immer? Haben wir immer gesagt?“*

„Nun, Ihr habt nicht immer gesagt, aber das passiert doch immer, oder?“ Na ja, es passiert immer dann, wenn der Compiler genügend Informationen hat und es notwendig ist. Und schon wieder hören wir jemanden aus der letzten Reihe bemerken: *„Wie, ist der Compiler manchmal blind?“* Ja, so ungefähr!

Das Problem kommt genau dann zum Tragen, wenn wir Funktionen benutzen die der Compiler nicht kennt. Nehmen wir beispielsweise die Funktion `double sin( double x)`, die ein Argument vom Typ `double` erwartet und ebenso einen Wert vom Typ `double` zurück gibt. Würden wir jetzt die Funktion wie folgt aufrufen `sin( 2 )` ginge es schief (siehe aber nächsten Absatz). Warum? Ganz einfach. Die Funktion `sin()` denkt, dass da ein Argument vom Typ `double` kommt. Also auf meinem Rechner acht Byte in der für `double` spezifischen Kodierung. Aber was kommt? Ja, es kommen bei mir vier Bytes in einer `int` Kodierung, bei der nicht nur vier Bytes fehlen sondern die Zahlen auch noch ganz anders kodiert werden. Das muss schief gehen! Der Grund hierfür liegt darin, dass der Compiler bei einer für ihn unbekannten (nicht deklarierten) Funktionen annimmt, dass sie vom Typ `int` ist und nur Parameter vom Typ `int` hat.

Früher ging das Verwenden nicht deklarierter Funktionen auch immer wieder schief, was

oft eine zeit-und nervenraubende Fehlersuche nach sich zog. Die gute Nachricht ist, dass durch den Standard C99 [13] dem Compiler die Standardfunktionen bekannt sind und er weiß, was er zu tun hat. Zusätzlich gibt er zu unserer Kenntnisnahme eine entsprechende Warnung aus. Aber wenn wir unsere eigenen Funktionen schreiben, kann dieses Problem weiterhin auftreten, worauf wir in den Kapiteln 55 nochmals zurück kommen.

Die aus der Verwendung nicht deklarierter Funktionen resultierenden Probleme können uns aber auch begegnen, wenn wir die `char`-Klassifikationen wie beispielsweise `isalpha()` (siehe Kapitel 30) mit den falschen Typen verwenden *und* vergessen, die Datei `ctype` mittels `#include <ctype.h>` einzubinden.

„Liegt es jetzt nur an dem `#include`*“?* Ja und nein ;-) In diesen Include-Dateien stehen diverse Funktionsdefinitionen wie beispielsweise `double cos( double x );` durch die das genaue typmäßige Aussehen der Funktion[1] dem Compiler bekannt gemacht wird. Anschließend kann der Compiler die implizite Typanpassung selbstständig durchführen; er weiß ja nun, wie es sein soll und was zu tun ist.

Und da wir gerade so schön dabei sind, gehen wir in den nächsten beiden Kapiteln etwas näher auf den Complier `gcc` ein; ihr wollt ja wissen, wie alles funktioniert ;-) Wir fangen mit dem Präprozessor an und besprechen anschließend den Rest des Compilers.

[1] Wer jetzt noch mehr wissen möchte, schaut unter Linux einfach mal in die folgende Include-Datei: `/usr/include/bits/mathcalls.h` Dort ist alles ein bisschen kompliziert, aber mit etwas Geduld bekommt man es heraus.

Kapitel 38

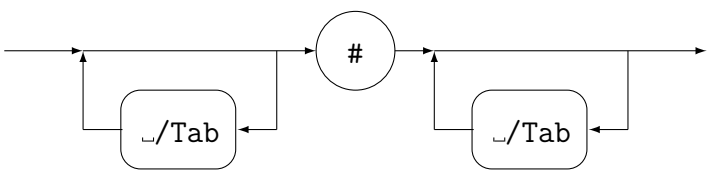
Der Präprozessor cpp

Um es gleich klar zu stellen: Der C-Präprozessor ist *kein* (!) C-Compiler. *„Wie bitte?“* höre ich es aus der letzten Reihe fragen, *„Der C-Präprozessor ist kein C-Compiler? Wieso heißt er denn dann so?“* Nun, der C-Präprozessor ist dem eigentlichen Compiler vorgeschaltet und führt einige Textersetzungen durch, ohne sich um das C-Programm als solches zu kümmern. *„Und wie muss ich mir das vorstellen?“* Das ist ungefähr so wie im täglichen Leben. Wenn Frau Müller und Herr Schulze heiraten, wird voraussichtlich einer der beiden seinen Namen ändern; eine typische Aufgabe für den Präprozessor. Der Präprozessor ist ein einfaches Textersetzungssystem, dass der eigentliche C-Compiler für unsere Bequemlichkeit mit aufruft. Mit ihm kann man ganze tolle Sachen machen, wie z.B. Webseiten auf einfache Weise schön konsistent halten. Aber das geht hier zu weit. In diesem Kapitel beschäftigen wir uns der Reihe nach kurz mit den für uns wesentlichen Präprozessor-Direktiven `#include`, `#define` und `#ifdef`. Doch zuvor ein paar grundsätzliche Bemerkungen.

38.1 Generelles zu den Präprozessor-Direktiven

Die Präprozessor-Direktiven sind die Zeilen, die mit einem Doppelkreuz `#` anfangen. Also beispielsweise `#include <stdio.h>`, wie wir es schon des Öfteren hatten. Es ist tatsächlich so, dass die Zeilen mit den Präprozessor-Direktiven mit einem Doppelkreuz anfangen müssen. Man kann aber vor und nach dem Doppelkreuz beliebig viele Leer- und Tabulatorzeichen einfügen. Entsprechend sind auch ␣␣`#include <stdio.h>` und ␣`#`␣␣`include <stdio.h>` völlig korrekt. Formal lässt sich das wie folgt ausdrücken:

`#`-Präfix



Hinzu kommt noch, dass *vor* der Bearbeitung der Präprozessor-Direktiven etwaige Kommentare entfernt werden. Entsprechend wären auch folgende Varianten korrekt, von denen wir aber ausdrücklich nur die ersten beiden Varianten empfehlen!

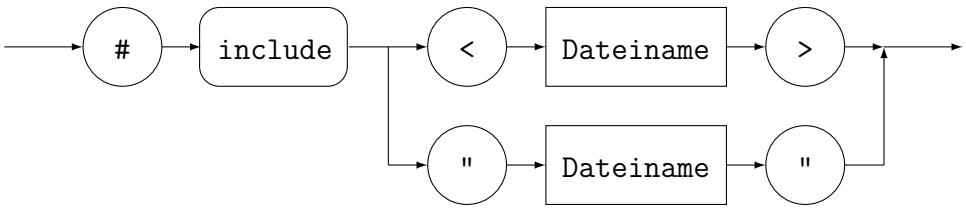
```
1 #include <stdio.h>                          // der Normalfall
2    #include <stdio.h>                       // siehe oben
3 #     include <stdio.h>                     // siehe oben
4 /* Kommentar am Anfang */ #include <stdio.h>
5   #  /* Kommentar in der Mitte */ include <stdio.h>
```

Allen Direktiven ist gemein, dass sie nach der Bearbeitung durch den Präprozessor nicht mehr vorhanden sind. Ferner müssen die Direktiven in einer Zeile abgeschlossen sein. Sollte der Platz nicht reichen, kann man am Zeilenende ein Backslash \ setzen und in der nächsten Zeile weiterschreiben. Der Präprozessor fügt solche Zeilen zu einer zusammen und entfernt natürlich die beiden Zeichen Backslash und das Zeilenende.

38.2 Die `#include`-Direktive

Die `#include`-Direktive haben wir schon zu genüge verwendet. Der Präprozessor ersetzt die Direktive durch die angegebene Datei, egal, um was für Inhalte es sich handelt. Dabei kann die Datei in zwei Formen angegeben werden, `<datei>` und `"datei"`. In der ersten Variante schaut der Präprozessor nur in den ihm bekannten Verzeichnissen nach (unter Linux beispielsweise `/usr/include`), ob dort die angegebene Datei zu finden ist. In der zweiten Variante fängt er mit der Suche im aktuellen Verzeichnis an und sucht ggf. in den oben erwähnten Verzeichnissen weiter. Das zugehörige Syntaxdiagramm sieht wie folgt aus:

`#include`-Direktive



Dem Syntaxdiagramm kann man ferner entnehmen, dass pro Zeile nur eine Datei angegeben werden darf. Zur Illustration präsentieren wir folgendes kleines, hypothetisches Beispiel: Wir nehmen an, dass wir die zwei `.h`-Dateien namens `oma.h` und `opa.h` in unserem aktuellen Verzeichnis haben:

oma.h

```
1 Vorname: Erna
2 Nachname: Alt
3 Alter: 95
4 Fitness: Super
5 Hobby: Ultimate Fighting
```

opa.h

```
1 Vorname: Erfried
2 Nachname: Alt
3 Alter: 97
4 Fitness: Ok
5 Hobby: Sport vorm Fernseher
```

Wir nehmen jetzt ferner an, dass wir die Datei `familie.txt` haben (linke Seite). In diesem Fall erzeugt der C-Präprozessor daraus die rechte Seite lediglich durch reine Textersetzung:

Kommando: `cpp familie.txt` (Ausgabe auf dem Bildschirm und Kommentare entfernt)

familie.txt ⇒⇒⇒⇒⇒⇒⇒⇒ **Bildschirmausgabe**

```
1 1. Meine Grosseltern
2
3 Meine Oma:
4 #include "oma.h"
5
6
7 Und mein Opa:
8 #include "opa.h"
```

```
1  1. Meine Grosseltern
2
3  Meine Oma:
4  Vorname: Erna
5  Nachname: Alt
6  Alter: 95
7  Fitness: Super
8  Hobby: Ultimate Fighting
9
10 Und mein Opa:
11 Vorname: Erfried
12 Nachname: Alt
13 Alter: 97
14 Fitness: Ok
15 Hobby: Sport vorm Fernseher
```

Verblüffend einfach, wenn man es sich recht überlegt.

38.3 Die `#define`-Direktive

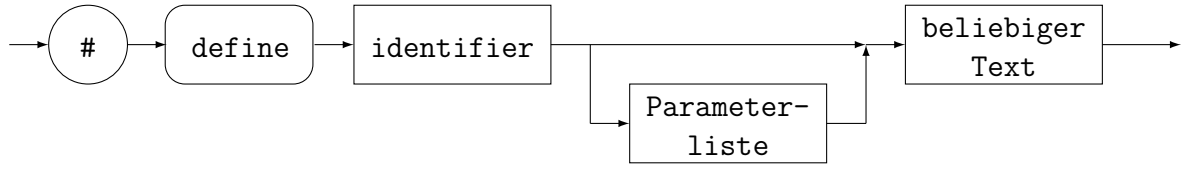
Mit der `#define`-Direktive kann man zwei Dinge machen. Zum Einen kann man „Label"s definieren, die später im Quelltext ersetzt werden. Beispiele sind `#define N 10` bzw. `#define NAME James Bond`. Überall dort, wo im folgenden Quelltext `N` bzw. `NAME` als *eigenständige* Identifier auftauchen, werden die entsprechenden Ersetzungen durchgeführt.

Zum Zweiten kann man mit der `#define`-Direktive vollständige Makros mit Parametern definieren. Beispielsweise könnte man mittels `#define sum(a,b) (a)+(b)` überall im folgenden Quelltext eine Summenfunktion verwenden, die entsprechend ersetzt wird. Ok, das Beispiel ist nicht besonders praxisnah, dient aber der Illustration.

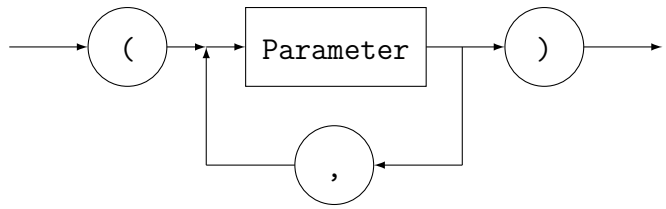
Die Verwendung von `#define`-Direktiven kann dazu beitragen, ein Programm lesbarer und vor allem änderungsfreundlicher zu machen: Zum einen können Ausdrücke mit „aussagekräftigen" Begriffen belegt werden und zum anderen können unter Umständen notwendige Änderungen auf einen Punkt konzentriert werden.

Das entsprechende Syntaxdiagramm sieht wie folgt aus:

#define-Direktive



#define-Parameterliste



Die `#define`-Direktiven bieten keine neuen Sprachkonstrukte oder dergleichen an, erhöhen aber die Änderbarkeit und Wartbarkeit eines Programms enorm! Wenn man immer wiederkehrende Konstanten, Faktoren, Zeichenketten etc. an zentraler Stelle verwaltet, genügt oft eine einzige Änderung, um entsprechende Anpassungen vorzunehmen. Ferner ist dies „sicherer“ als Änderungen per Hand, denn bei letzteren vergisst man häufig eine Stelle oder ändert eine zu viel. Das folgende Beispiel illustriert das eben gesagte ein wenig (`#include <stdio.h>` haben wir weggelassen, da es das Ergebnis „unendlich“ aufblähen würde):

```
1 #define COUNT   10
2
3 int main( int argc, char **argv )
4     {
5         int i, sum;
6         for( sum = 0, i = 0; i < COUNT; i = i + 1 )
7             sum = sum + i;
8         printf( "COUNT= %d sum= %d\n", COUNT, sum );
9     }
```

In Zeile 1 haben wir eine Konstante namens `COUNT` definiert, die wir in den Zeilen 6 und 8 verwenden. Der Präprozessor macht genau das, was wir vermuten: er ersetzt das „Label“ `COUNT` durch die rechte Seite, also die Zahl 10, wie man an folgendem Ergebnis sehen kann:

```
1
2
3 int main( int argc, char **argv )
4     {
5         int i, sum;
6         for( sum = 0, i = 0; i < 10; i = i + 1 )
7             sum = sum + i;
8         printf( "COUNT= %d sum= %d\n", 10, sum );
9     }
```

Ferner sieht man am Ergebnis, dass der Präprozessor die Textersetzung *nicht* überall anwendet: konstante Zeichenketten (siehe `printf()` in Zeile 8) sind davon ausgenommen.

Das folgende Beispiel illustriert die Definition von Makros: Am Dateianfang definieren wir ein Makro `parallel( r1, r2)`, das den resultierenden Widerstand zweier parallel geschalteter Widerstände berechnet. Dieses Makro wenden wir anschließend drei Mal an:

parallel.c

```
 1 #define parallel( r1, r2 )      ((r1)*(r2)/((r1)+(r2)))
 2
 3 int main( int argc, char **argv )
 4     {
 5       double r, x1 = 2.0, x2 = 2.0;
 6       r = parallel( x1, x2 );
 7       printf( "x1=%3.1f x2=%3.1f x1//x2=%3.2f\n", x1, x2, r );
 8       r = parallel( 1.0 + 1.0, x2 );
 9       printf( "x1=%3.1f x2=%3.1f x1//x2=%3.2f\n", 2.0,x2, r );
10       r = parallel( parallel( 3.0, 3.0 ), 3.0 );
11       printf( "xi=3.0 x1//x2//x3=%3.2f\n", r );
12     }
```

Mittels des C-Präprozessors `cpp parallel.c` erhalten wir folgende Ausgabe:

```
 1
 2
 3 int main( int argc, char **argv )
 4     {
 5       double r, x1 = 2.0, x2 = 2.0;
 6       r = ((x1)*(x2)/((x1)+(x2)));
 7       printf( "x1=%3.1f x2=%3.1f x1//x2=%3.2f\n", x1, x2, r );
 8       r = ((1.0 + 1.0)*(x2)/((1.0 + 1.0)+(x2)));
 9       printf( "x1=%3.1f x2=%3.1f x1//x2=%3.2f\n", 2.0,x2, r );
10       r = ((((3.0)*(3.0)/((3.0)+(3.0))))*(3.0)/((((3.0)*(3.0)
11           /((3.0)+(3.0))))+(3.0)));
12       printf( "xi=3.0 x1//x2//x3=%3.2f\n", r );
13     }
```

Dieses Beispiel zeigt noch zwei weitere Dinge: Die Parameter des Makros können ganze Ausdrücke sein. In Zeile 8 bekommt der Parameter `r1` den Wert `1.0 + 1.0`, und in Zeile 10 verwenden wir sogar das eigene Makro noch einmal, was man übrigens *Rekursion* nennt.

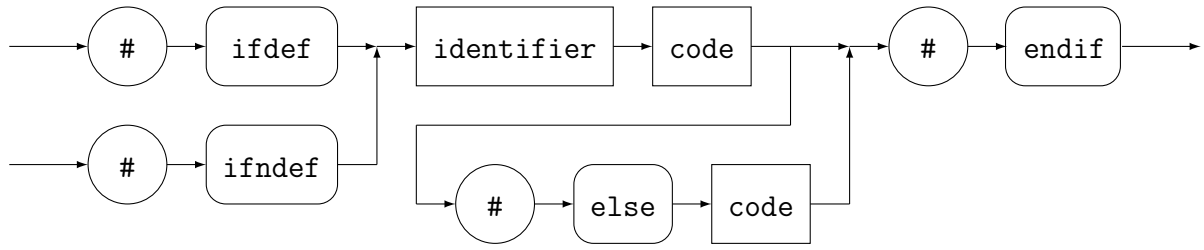
Manch einer mag sich noch über die Klammern um `(r1)` und `(r2)` wundern, denn das sieht im Endresultat doch etwas übertrieben aus. Ja, sie müssen aber sein, da der Präprozessor nur eine „stupide“ Textersetzung durchführt, in der Buchstabe für Buchstabe ersetzt wird. Hätten wir diese Klammern nicht und würden das Makro als `parallel( 1.0 + 1.0, x1)` aufrufen, käme aufgrund der Punkt- und Strichrechnung etwas falsches heraus. Aber was erzähle ich, probiert es doch einfach mal aus (auf Papier und dem Rechner).

Es bleibt noch zu erwähnen, dass auf der rechten Seite einer `#define`-Direktive beliebiger C-Code stehen darf. Aber das führt leicht zu Komplikationen und sollte dem fortgeschrittenen Programmierer vorbehalten sein. Also gehen wir auch nicht weiter darauf ein.

38.4 Die `#ifdef`-Direktive

Mittels der `#ifdef`-Direktive, die man sich als „falls definiert“ vorstellen kann, werden in Abhängigkeit vorheriger Definitionen unterschiedliche Aktionen ausgelöst. Eine derartige Direktive besteht aus einem `then`-Teil, einem optionalen `#else`-Teil und wird durch ein `#endif` abgeschlossen. Alternativ kann man auch `#ifndef` („falls nicht definiert“) benutzen, wobei sich dann die beiden Anweisungsteile vertauschen. Nach dem `#ifdef`-Teil muss noch ein Identifier folgen. Und je nach dem, ob er definiert ist oder nicht, werden die Zeilen des entsprechenden Teils ausgeführt. Das formale Syntaxdiagramm sieht wie folgt aus:

`#idef-Direktive`



Es bleibt noch zu erwähnen, dass der `Code`-Block erst in der nächsten Zeile anfangen darf und dieser auch leer sein kann. Zur Illustration präsentieren wir im Folgenden ein halbwegs sinnvolles Beispiel, dass beim Fehlerfinden während des Programmierens helfen soll:

```
 1 #define _DEBUG
 2
 3 #ifdef _DEBUG
 4    #define _int_debug( var, val ) \
 5            printf( "debug: var=%s value=%d\n", var, val )
 6 #else
 7    #define _int_debug( var, val )
 8 #endif
 9
10 int main( int argc, char **argv )
11     {
12        int i;
13        for( i = 0; i < 2; i++ )
14        {
15           _int_debug( "i", i );
16           printf( "huhu, die %dte\n", i );
17     }  }
```

Für den Fall, dass das „Label“ `_DEBUG` gesetzt ist, wird in den Zeilen 4 und 5 ein Makro namens `_int_debug()` definiert, das den Namen und den Wert einer Variablen ausgibt. Die Beispielverwendung befindet sich in Zeile 15. Und tatsächlich wird folgendes auf dem Bildschirm angezeigt:

```
1 debug: var=i value=0
2 huhu, die 0te
3 debug: var=i value=1
4 huhu, die 1te
```

So, nach dem Beispiel noch ein paar Erläuterungen:

1. Für den Fall, dass wir diese Debug-Informationen nicht haben wollen, können wir das `#define` in der ersten Zeile einfach auskommentieren: `//#define _DEBUG`.

   In diesem Fall verschwindet das Makro `_int_debug()`, da keine rechte Seite der Definition vorhanden ist.

2. Bei der Abfrage `#ifdef _DEBUG` macht es nichts, dass dieses „Label“ keinen Wert hat; durch das `#define _DEBUG` ist es definiert, auch wenn der Inhalt leer ist.

3. Der Übersichtlichkeit halber kann man die ganzen Präprozessor-Direktiven einrücken, wie man deutlich in den Zeilen 5 und 7 sieht.

4. In den einzelnen Teilen könnte natürlich auch mehr als eine Anweisung stehen, wenn man denn mehrere benötigt. Dies können auch weitere `#ifdef`-Direktiven sein.

Und wofür braucht man das Ganze? Nun, ein Anwendungsfall ist das gezeigte Debugging eines Programms. Sollte das Programm irgendwann funktionieren, kann man alle Debugging-Anweisungen durch das Auskommentieren einer einzelnen Zeile entfernen. Schon ziemlich praktisch.

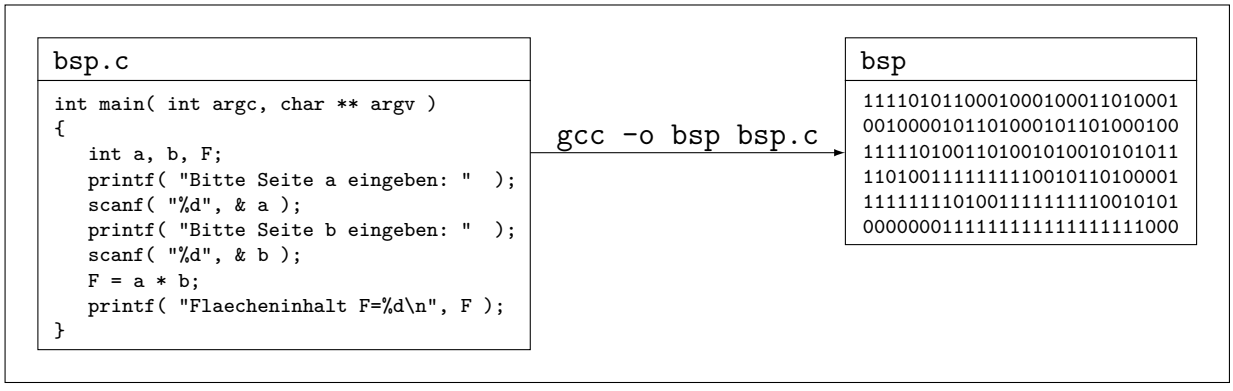
Ferner kann man diese Direktive sehr gut beim Schreiben eigener `.h`-Dateien einsetzen. Aber dazu kommen wir erst in Kapitel 55.

Kapitel 39

Der Compiler und seine Arbeitsschritte

Vom Compiler (Übersetzer) haben wir schon in Kapitel 8 gehört. Er ist dafür zuständig, ein in einer Programmiersprache wie C geschriebenes Programm in ein von der CPU ausführbares Maschinenprogramm zu übersetzen. Dies geschieht für gewöhnlich in mehreren, aufeinanderfolgenden Arbeitsschritten. Im Falle von C-Programmen erledigt der `gcc` dies in vier größeren Schritten. Die Komponenten heißen: Präprozessor, eigentlicher Übersetzer, Assembler und Linker. Bei jedem dieser Schritte können Fehler auftreten, die vom Compiler ausgegeben werden. Doch wenn alles gut gelaufen ist, kommt zum Schluss ein lauffähiges Programm heraus.

Für den Programmieranfänger ist wichtig zu verstehen, dass diese vier Schritte auch dann nacheinander ausgeführt werden, wenn man den Compiler mittels `gcc` aufruft. Ebenso verfahren die integrierten Entwicklungssysteme nach diesem Schema, wenn man die entsprechende Funktionstaste wie beispielsweise `F5` betätigt. Zur Wiederholung hier nochmals das hier relevante Teilbild aus Kapitel 8:

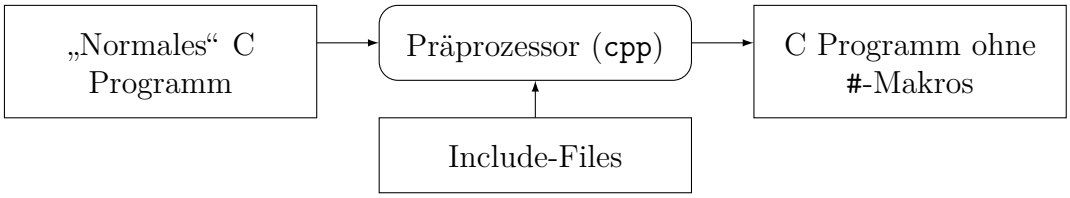


39.1 Der Aufruf des Compilers `gcc`

Der Aufruf des Compilers `gcc datei.c` führt nacheinander alle notwendigen Schritte aus und speichert das entsprechende lauffähige Programm in der Datei `a.out` (Linux) bzw. `a.exe` (Windows), das anschließend durch `a.out` (Linux), `./a.out` (Linux) bzw. `a.exe` (Windows) ausgeführt werden kann. Anfänglich werden sich unsere Programme immer in *einer* Datei befinden; die Aufteilung in mehrere Dateien besprechen wir erst in Kapitel 55.

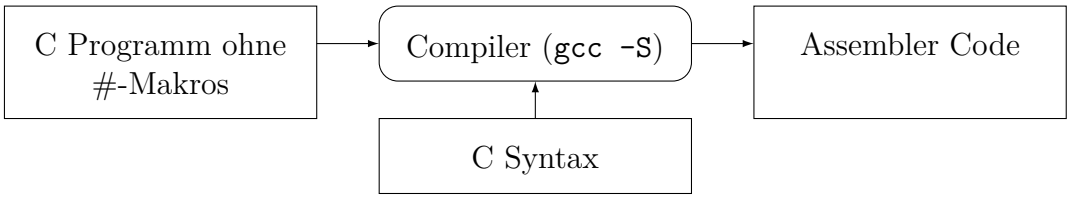
39.2 Der Präprozessor

Den Präprozessor haben wir bereits im vorherigen Kapitel ausführlich behandelt. Zur Erinnerung: Er bearbeitet die #-Direktiven wie beispielsweise `#include`, `#define` und `#ifdef`. Seine Funktion besteht lediglich im Ersetzen von Texten. Eine grafische Darstellung dieses Vorgangs könnte wie folgt aussehen:



39.3 Der eigentliche Compiler

Das Ergebnis des Präprozessors war eine Datei, die nur noch Anweisungen der Programmiersprache C enthält. Der eigentliche Compiler wandelt nun die C-Anweisungen in eine Form um, die man auch Assembler Code nennt. Dabei bedient er sich der Definitionen der Sprachbeschreibung, die die gültige Syntax aller Anweisungen definieren:



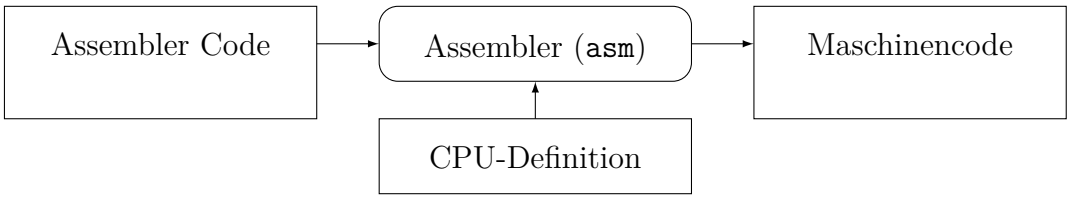
In diesem Schritt geschieht der eigentliche Übersetzungsvorgang. Die aus CPU-Sicht abstrakten C-Anweisungen werden so umgesetzt, dass sie die elementaren Möglichkeiten der CPU widerspiegeln. Mit anderen Worten, das Resultat dieses Übersetzungsschrittes, also der Assembler Code, ist schon fast von der CPU ausführbar. In diesem Übersetzungsschritt wird aber auch schon der benötigte Platz und Speicherbereich für alle Variablen und auszuführenden Anweisungen vorreserviert. Ferner wird in diesem Schritt noch die eine oder andere Optimierung automatisch durchgeführt. Bei diesen Optimierungen berücksichtigt

der Compiler die Möglichkeiten und Spezifika des zukünftigen Zielrechners. Das Ergebnis dieses Übersetzungsschrittes wird üblicherweise in Dateien abgelegt, die mit `.s` enden.

Bei etwas tiefergehendem Nachdenken sollte einem hier folgender Sachverhalt offenbar werden: Bei x verschiedenen Programmiersprachen und y unterschiedlichen CPUs benötigt man im Grunde genommen $x \times y$ verschiedene Compiler. Diese Zahl wird dadurch erheblich reduziert, dass sich die Prozessoren stark ähneln und auch einige Programmiersprachen ineinander überführt werden können. Eine vertiefte Diskussion würde hier aber den Rahmen deutlich sprengen.

39.4 Der Assembler

Der Assembler-Code, der im vorherigen Schritt generiert wurde, besteht aus einzelnen Assembler-Befehlen. Diese Befehle sind immer noch im Klartext gegeben, sind aber bereits sehr *low level*, da es für jeden Assembler-Befehl auch einen oder mehrere entsprechende CPU-Instruktionen gibt, die direkt von der Hardware ausgeführt werden können. Der Assembler ist nun ein Programm, das die in Klarschrift gegebenen Befehle, auch mnemotechnische Codes genannt, in die korresponierenden 0/1-Kombinationen umsetzt. Das Resultat wird üblicherweise in einer Datei abgelegt, die mit `.o` endet.



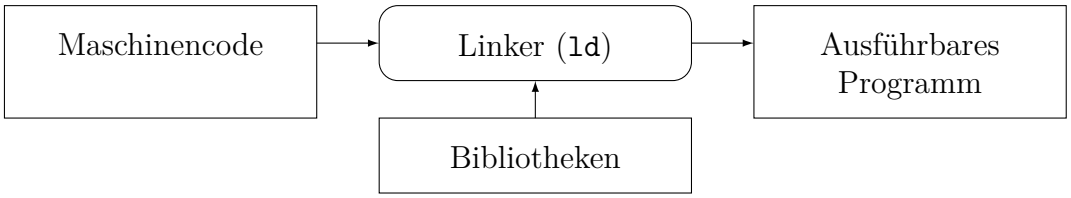
Das Resultat besteht im Wesentlichen aus den folgenden drei Teilen:

1. Dem Programmcode, der seinerseits aus den Hardware-Instruktionen besteht und den zugehörigen Daten, die die Konstanten sind.
2. Einer Liste aller Objekte, die innerhalb dieses assemblierten Programmstücks *realisiert* sind. Das Wort Objekt ist etwas hochgestochen aber dennoch üblich und meint die „globalen“ Variablen und Funktionen. Ein Beispiel könnte die Funktion `main()` sein, sofern sie im Quelltext vorhanden ist.
3. Einer Liste aller Objekte, die innerhalb dieses assemblierten Programmstücks *noch nicht* realisiert sind, also *noch benötigt* werden. Ein Beispiel hierfür könnte die Funktion `printf()` sein, sofern sie verwendet wird, denn diese implementiert man ja nicht selbst, sondern benutzt sie aus der Standardbibliothek.

Im Grundegenommen ist das Resultat des Assemblers lauffähig, wenn da nicht die beiden Listen (Punkt 2 und 3) wären...

39.5 Der Linker

Wie eben gesagt, wären die zuvor generierten `.o`-Dateien eigentlich schon lauffähig. Was jetzt noch fehlt, sind die Bibliotheken, in denen sich beispielsweise die Ausgabeanweisung `printf()` befindet. Mittels der Anweisung `#include <stdio.h>` macht man dem Compiler nur klar, dass es *irgendwo* eine Ausgabeanweisung namens `printf()` gibt. Wo sich aber der zugehörige Maschinencode befindet, weiss er noch *nicht*. Diese Dateien nennt man Bibliotheken, die jetzt noch zum eigenen Programm hinzugefügt werden müssen.



Für diesen letzten Schritt nimmt der Linker die zuvor generierten `.o`-Dateien, packt alle zusammen, sortiert diese nach seinen eigenen Richtlinien und löst vor allem die oben genannten Listen auf. Das heißt, der Linker schaut zuerst, ob ein und das selbe Objekt in zwei oder mehr Listen *angeboten* wird. Sollte dies der Fall sein, bricht der Linker ab, da es zu Mehrdeutigkeiten kommen würde. Anschließend schaut der Linker, ob er für jedes *benötigte* Objekt in einer der anderen Dateien eine Realisierung findet und trägt die entsprechenden Adressen ein. Anschließend wird das gesuchte Objekt aus der Liste der benötigten Objekte gestrichen. Am Ende dieses Prozesses muss in allen Dateien die Liste der benötigten Objekte leer sein. Ist dies der Fall, ist die Aufgabe erfolgreich gelöst und man erhält eine ausführbare Datei `a.out`.

Sollte eine der Listen *nicht* leer sein, gibt der Linker eine Fehlermeldung der Art „unresolved external ...“ aus. Dies ist beispielsweise der Fall, wenn man eine Funktion wie sin() verwendet, aber beim Linken die entsprechende Bibliothek nicht mittels `-lm` angibt. Der Linker weiß nämlich nicht von sich aus, welche Bibliotheken benötigt werden und wo er sie finden kann. Dies „sagt“ man ihm mittels der Option `-l<bibliotheksname>`. Wie eine Bibliothek heisst, erfährt man beispielsweise unter Linux/Unix auf der entsprechenden *man page*. Eine Ausnahme bildet die Standardbibliothek `libc`. Diese wird *immer* dazugebunden und enthält beispielsweise alle Ein-/Ausgabefunktionen.

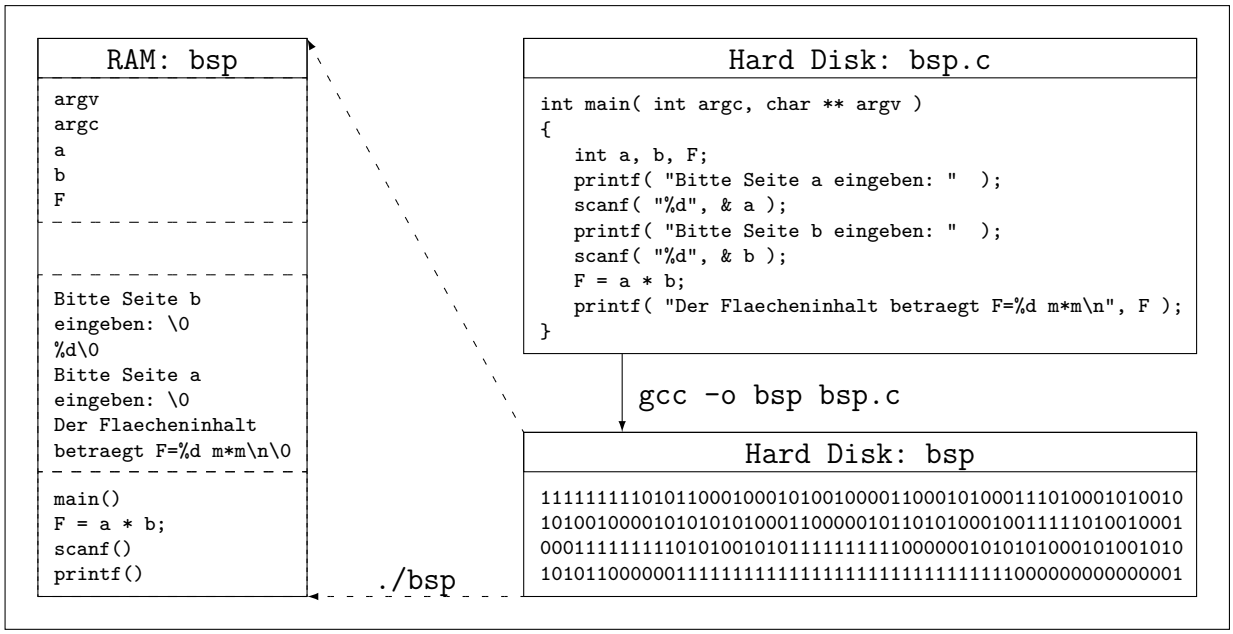
39.6 Zusammenfassung

Der Compiler verrichtet seine Arbeit im Wesentlichen in vier aufeinanderfolgenden Schritten. Als Programmieranfänger schaut man sich normalerweise die einzelnen Schritte nicht so genau an, sondern ruft direkt den Compiler mit allen Teilschritten auf: `gcc datei.c` bzw. `gcc -o programm datei.c`. In diesem Falle werden auch die Zwischenergebnisse nicht nachhaltig abgelegt, sondern unmittelbar nach ihrer Verwendung wieder gelöscht.

Kapitel 40

Die Speicherorganisation durch den Compiler

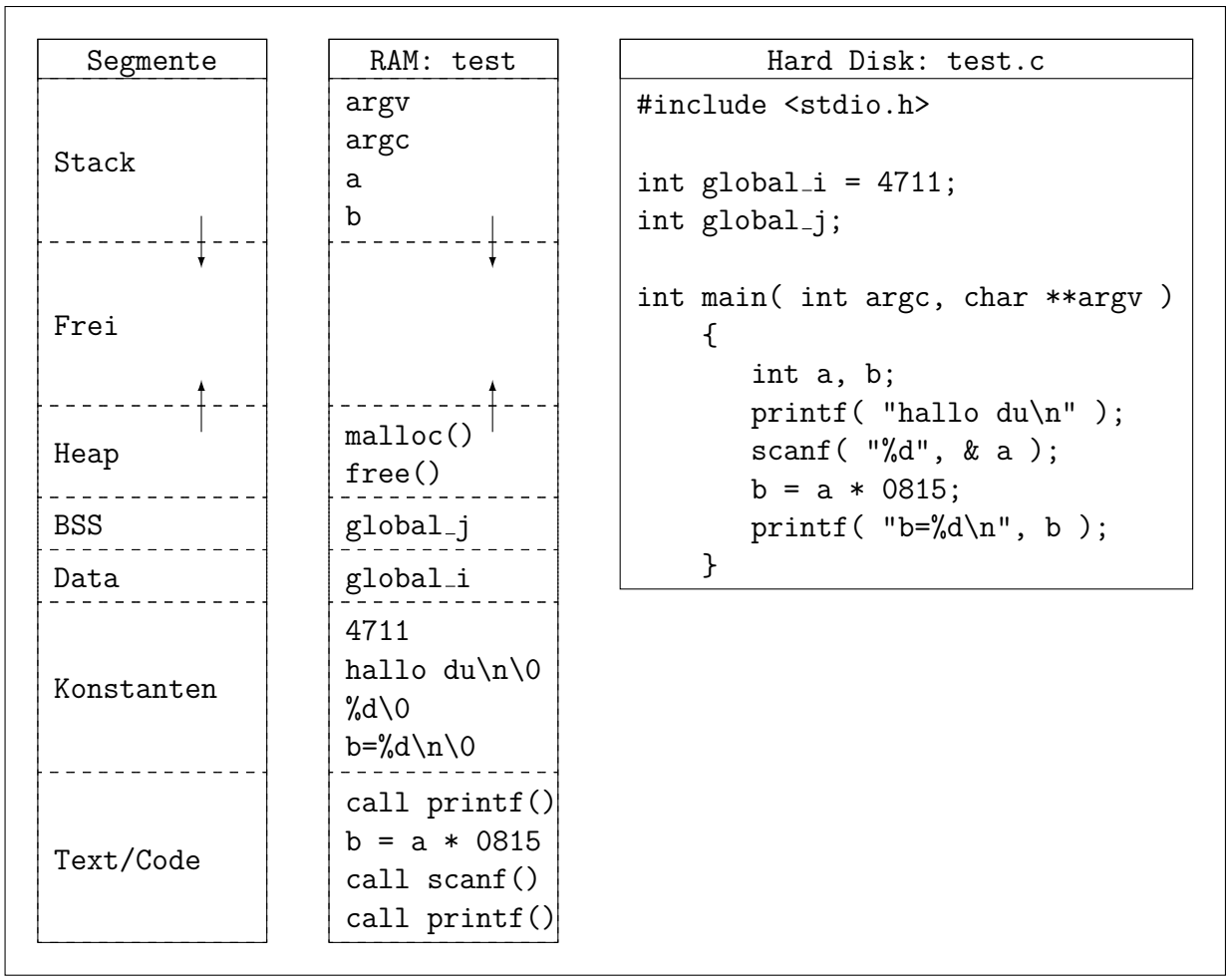
„Ordnung ist das halbe Leben," heißt es so schön. Aber wenn man das halbe Leben lang Ordnung halten muss, ist es auch nicht so schön. Na ja, was soll's ... Bereits in Kapitel 8 haben wir auf Seite 34 versucht, Euch einen ersten Eindruck davon zu vermitteln, wie der Compiler die einzelnen Anweisungen und Daten voneinander trennt und in verschiedene Segmente aufteilt. Zur Wiederholung und als Einstieg haben wir dieses Bild nochmal hier unten abgebildet. Diese segmentartige Organisation macht es dem Compiler einfacher, ein C-Programm zu erstellen, und ermöglicht es der Hardware, wirkungsvolle Schutzmechanismen zur Verfügung zu stellen. Insbesondere kann die Hardware darauf achten, dass keine der Anweisungen versehentlich modifiziert wird.



Unten auf dieser Seite haben wir das eingangs gezeigte Bild nebst des Testprogramms vervollständigt; es sind jetzt alle Segmente abgebildet, die aus Sicht eines Anwenderprogrammierers erst mal wichtig sind. Im Detail sind die Dinge – wie sollte es auch anders sein – doch noch ein wenig komplizierter. Aber diese Dinge wären eher Gegenstand einer Vorlesung aus dem Informatik-Hauptstudium. Sollte doch jemand tiefergehendes Interesse haben, keine Scheu haben und einfach vorbei kommen oder uns nach der Vorlesung oder in den Übungen ansprechen.

Der Eine oder Andere wird sich vielleicht fragen, warum wir hier so abstrakt über die Speichersegmente reden. Ganz einfach, das Verständnis der Speicherorganisation durch den Compiler hilft uns nachher, besser zu verstehen, wie die fortgeschrittenen Sprachkonstrukte wirklich funktionieren. Ferner werden wir bei der Vorstellung diese Sprachkonstrukte nicht wieder auf die Grundlagen der Speicherverwaltung eingehen, was vom eigentlichen Thema nur ablenken würde. Im Folgenden besprechen wir jedes einzelne Segment.

Speichersegmente am Beispiel test bzw. test.c



40.1 Text- bzw. Code-Segment

Traditionell befindet sich das Textsegment, das auch Code-Segment genannt wird, im unteren Speicherbereich. Hier befinden sich alle Anweisungen, also alle Zuweisungen, Funktionsaufrufe, Fallunterscheidungen, Schleifen, Berechnungen usw. Mit anderen Worten: Hier sitzt das eigentliche Maschinenprogramm und darauf greift nur der *Program Counter* (PC) zu, wie wir es bereits in Kapitel 36 besprochen haben.

Der tiefere Sinn eines Maschinenprogramms ist, dass es ausgeführt wird. Klar. Und da Maschinenprogramme zum eigenen Schutz während der Ausführungsphase nicht veränderbar sein sollten, können hier Betriebssystem und Hardware gewisse Schutzmechanismen einbauen. Sollte man beispielsweise (bedingt durch einen Programmierfehler) hier etwas hineinschreiben wollen, merkt das die Hardware und bringt das Betriebssystem dazu, das Programm abzubrechen. Diese Abbruch ist immer noch besser als ein unkontrolliertes Weiterlaufen des fehlerhaften Programms. Ferner unterbinden Hardware und Betriebssystem jeden Versuch, irgendwelche Daten als Code zu interpretieren und ausführen zu wollen.

40.2 Konstanten-Segment

Hier stehen alle Konstanten, die als solche weiter verwendet werden sollen und/oder eine gewisse Speichergröße überschreiten. Vornehmlich sind das mal alle Zeichenketten, die wir beispielsweise in unseren Ausgabeanweisungen (`printf()`) sowie Eingabeanweisungen (`scanf()`) verwenden.

„Aber im Programm stehen doch die diversen Zeichenketten in unterschiedlichen Ausgabeanweisungen... Kommt da das Programm nicht durcheinander?“ Nein, kommt es nicht. Aber die Frage ist natürlich sehr gut. Der Compiler sammelt alle Zeichenketten und packt sie in dieses Segment. Im Text-Segment stehen dann nur die entsprechenden Funktionsaufrufe, denen die Adressen der jeweiligen Zeichketten innerhalb dieses Segmentes übergeben werden, sodass die Ausgabeanweisung weiß, wo sie ihren Text findet; ja, die Ausgabeanweisung öffnet die einzelnen Schubladen und holt sich Buchstabe für Buchstabe selbst aus dem Speicher, um den Text auszugeben.

Diese Trennung von Textsegment und den Konstanten hat folgende Gründe:

1. Man kann den Schutz erhöhen, denn der Program Counter greift nicht auf dieses Segment zu.
2. Sollte ein Programm versuchen, seine eigenen Ausgabetexte durch fehlerhafte Anweisungen zu verändern, wird es abgebrochen.
3. Irgendwie müssen diese Konstanten, wie auch das Text-Segment in den Arbeitsspeicher. Dazu wird auf dem PC alles so wie es sein soll, Zeichen für Zeichen in die Datei (die `.o`-Datei) geschrieben. Bei einem Mikrocontroller oder dergleichen stehen solche Dinge im ROM, EPROM, EEPROM, FLASH, Boot-PROM etc. fest drin, sodass sie

auch beim erneuten Einschalten wieder verfügbar sind.

Ein wenig Aufmerksamkeit verdienen noch die beiden Konstanten `0815`[1] und `4711`. Die erste der beiden Zahlen ist im Textsegment untergebracht, die zweite hier oben im Konstanten-Segment. Der Grund hierfür liegt in der Verwendung der zweiten Zahl zur Initialisierung der globalen Variablen in Zeile 3 des Beispielprogramms. Mehr hierzu im nächsten Abschnitt.

40.3 Data-Segment

Das Data-Segment haben wir bisher noch nie erwähnt. Es beherbergt globale Variablen, die wir bisher auch noch nicht erwähnt bzw. verwendet haben.

Wie man im Beispielprogramm gut erkennen kann, sind globale Variablen ganz normale Variablen, die aber *außerhalb* der `main()`-Funktion und außerhalb jeder beliebigen anderen Funktion deklariert werden. Programmieranfänger verwenden diese Variablen sehr gerne, weil man „überall“ auf sie zugreifen kann. Aber durch sie werden Programme unleserlich und auf Dauer nicht mehr nachvollziehbar. Aus Sicht des Software Engineerings sind diese Variablen auch ein echtes *no go*, weshalb wir sie auch bisher in keinem unserer Beispielprogramme verwendet haben.

Ein wesentliches Merkmal dieser Variablen ist, dass man ihnen von Anfang an einen Wert geben kann. Aber bei der Aussage kommen (hoffentlich) gleich ein paar Fragen auf: *Wie kann das sein, denn die Zuweisung steht in keiner einzigen Funktion? Geschieht diese Zuweisung bereits vor dem Start des „Hauptprogramms“* ***`main()`****? Falls ja, wer macht dies?* Die Antwort ist ein klares ja!

Ja, diese Zuweisungen werden vor der ersten Anweisung der `main()`-Funktion abgearbeitet. Und der sich dahinter befindende Mechanismus ist recht einfach: Das Betriebssystem sorgt gar nicht dafür, dass die `main()`-Funktion zuerst aufgerufen wird. Zu Anfang wird eine Funktion namens `_init()` zur Ausführung gebracht. Diese sorgt für die Initialisierungen (und diejenigen Variablen, die wir im nächsten Abschnitt besprechen) und ruft dann `main()` auf. Anschließend wartet die `_init()`-Funktion darauf, dass `main()` fertig ist, ebenso wie beispielsweise `main()` darauf wartet, dass die Ausgabe `printf()` fertig wird.

Die Initialisierung der globalen Variablen ist verblüffend einfach. Der Compiler sammelt nicht alle Initialisierungsanweisungen, sondern packt alle Konstanten zusammen (in unserem Fall `4711`) und packt sie in genau der selben Reihenfolge in das Konstanten-Segment wie er die globalen Variablen in das Data-Segment packt. Für die Initialisierung kopiert er einfach alle Bytes vom entsprechenden Teil des Konstanten-Segments in das Data-Segment. Klar? Falls nicht, vorbeikommen und fragen.

[1] Ja, diese Konstante kann es in C nicht geben, wissen wir. Aber warum eigentlich nicht? Und wir bleiben bei dieser Zahl, da sie in der Informatik eine dieser Standardzahlen ist genauso wie die Zahl 4711.

40.4 BSS-Segment

Das BSS-Segment (uninitialized data) ist das zweite Segement, das globale Variablen beherbergt. Also gilt hier auch erst einmal alles, was wir vorher über die Verwendung globaler Variablen gesagt haben. Im Unteschied zu vorher ist es so, dass diese Variablen keine Initialwerte bekommen. Keine? Nun, keine besonderen. Per Definition im C99 Standard [13] werden alle diese Variablen auf null gesetzt. Dies ist die zweite, oben erwähnte Initialisierung, die die Funktion `_init()` für uns erledigt. Mit anderen Worten: Diese uninitialisierten, globalen Variablen haben anfänglich garantiert den Wert null.

Die Realisierung des Null-Setzens ist sehr einfach. Der Compiler (speziell der Linker) weiß am Ende, wie groß dieses Segment ist (wie viele Schubladen der Speicher hier hat) und kann alle Bytes mittels einer Schleife bequem auf null setzen.

40.5 Heap-Segment

In das Heap-Segment kommen alle Variablen, die wir uns *nachträglich* besorgen. Ja, richtig gelesen, man kann sich auch im Nachhinein während der Programmlaufzeit neue Variablen besorgen. Dies ist aber eher etwas schwierig und mit dem `malloc()/free()`-Mechanismus verbunden, den wir erst in Skriptteil VII erklären. Übrigens, „Heap“ ist englisch und bedeutet Haufen, und dieser wächst je nach Bedarf nach oben.

40.6 Stack-Segment

Das Stack-Segment, oder einfach der Stack, ist dasjenige Segment, in das der Compiler die Variablen aller Funktionen packt. Dieser Vorgang ist im Beispielprogramm anhand der Variablen `a`, `b`, `argc` und `argv` illustriert.

Was heißt eigentlich „Stack“ und wie funktioniert dieser? Stack bedeutet Stapel bzw. Stapelspeicher. Die Funktionsweise entspricht einem Zettelstapel auf dem Schreibtisch: Man legt neue Blätter immer oben auf und nimmt sie auch von oben wieder herunter. Die Blätter ganz unten müssen also am längsten warten. Diesen Vorgang nennt man auch LIFO für *last in, first out*: der zuletzt oben abgelegte Zettel wird auch als erster wieder entfernt.

Diese Konzept eignet sich hervorragend für die Realisierung von Funktionen: wie in der Mathematik und allen unseren Beispielen wird eine neu augerufene Funktion erst vollständig abgearbeitet, bis die aufrufende Funktion zur nächsten Anweisung übergehen kann. Diesen Ablauf haben wir bereits oben am Beispiel von `main()` und den Ein-/Ausgabeanweisungen `scanf()` und `printf()` erläutert. Einfach ausgedrückt: jede neu aufgerufene Funktion packt einen neuen Zettel auf den Stapel und entfernt ihn wieder, wenn sie fertig geworden ist. Zwischendurch wächst der Stack, aber am Ende des Programms ist er leer.

Da dieses Stack-Konzept für die Abarbeitung von Funktionen so elementar ist, hat jeder Prozessor hierfür ein eigenes Register, das *Stack Pointer* (SP) genannt wird. Dieses

Hardware-Register zeigt immer an das untere Ende des Stacks. Wird mehr Speicherplatz benötigt, geht der Stack Pointer entsprechend nach unten und später entsprechend wieder nach oben. Mehr Details hierzu präsentieren wir in den Kapiteln 44, 47 und 48.

40.7 Überlauf von Stack und Heap

Im Gegensatz zu den anderen Segmenten können Stack und Heap zur Laufzeit wachsen und schrumpfen. Normalerweise ist dies kein Problem, sodass man sich als Programmierer beruhigt zurücklehnen kann. Je nach Anwendung, Rechnersystem und Programmierfehlern kann es passieren, dass beide Segmente aneinanderstoßen. Die daraus resultierenden Konsequenzen hängen vom gewählten Rechnersystem ab. Ein PC beispielsweise erkennt diesen Fall durch Kooperation von Hardware und Betriebssystem und bricht das laufende Programm humorlos ab. Derartige Programmabstürze sind zwar nicht prickelnd, aber immerhin signalisieren sie diesen Fehler. Da auf den meisten Mikrokontrollern diese Hardwareunterstützung fehlt, werden derartige Fehler meist nicht erkannt. Als Folge werden einzelne Variablen überschrieben, was sich häufig in unerklärbaren Psi-Phänomenen äußert. Die Praxisrelevanz dieses Problem hängt primär von der zugrundeliegenden Hardware ab:

Kleinstsysteme/Mikrokontroller: Hier stehen oft nur 16-Bit breite Adressbusse zur Verfügung, sodass maximal 64 Kilo Bytes addressiert werden können. Oft stehen für Stack und Heap nur wenige Kilo Bytes zur Verfügung, sodass einigermaßen anspruchsvollen Anwendungen den Stack und Heap leicht überlaufen lassen.

32-Bit PC Architekturen: Eine 32-Bit PC Architektur erlaubt die Adressierung von 4 GB. Das Füllen eines derartig großen Arbeitsspeichers mit *sinnvollem* Material ist so schwer, dass Speicherüberläufe normalerweise nicht zu befürchten sind. Im Rahmen der Übungen werden Speicherüberläufe definitiv immer durch Programmierfehler wie Endlosrekursionen (Kapitel 48) und *Speicherleaks* (Skriptteil VII) verursacht.

64-Bit Architekturen: Mit einem 64-Bit Adressbus lassen sich $2^{64} = 2^{32} \times 2^{32}$ Bytes adressieren, was einem RAM von ungefähr vier Milliarden mal vier 4 GB entspricht. Bei unseren Anwendungen kann ein derartig großer Arbeitsspeicher in absehbarer Zeit nicht überlaufen. Einige Gründe lauten wie folgt:

1. Aneinander gereiht hätten die heutigen 8 GB Speicherriegel eine Länge von ungefähr 1 451 699 km. Zusammengepackt bräuchte man dafür etwa 200 Container.
2. Ein RAM mit 2^{64} Bytes könnte für etwa neun Milliarden Stunden bzw. eine knappe Million Jahre mp4-kodiertes Filmmaterial aufnehmen.
3. Alleine das Nullsetzen aller dieser Speicherzellen würde ungefähr vier Milliarden Sekunden dauern, was ungefähr 126 Jahren entspricht.

Nun, viel Spass mit diesem Speichermonstrum ;-) ...

Kapitel 41

Die Ein-/Ausgabe im Überblick

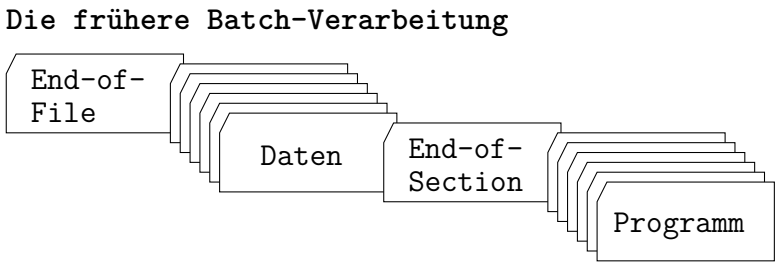
„Mittlerweile finde ich das Programmieren sogar ganz lustig. Aber was wirklich nervt, ist das Testen: manchmal funktioniert alles wie es soll, manchmal erlebe ich nur Endlosschleifen. Und meine Betreuer meinen dann manchmal noch, dass mein Programm eigentlich korrekt ist. Also, ich versteh's nicht, also nervt's!“ Ja, das können wir verstehen. Wir würden dir auch gerne helfen, aber das ist nicht einfach. *„Wieso isses nicht einfach?“* Prinzipiell kann man folgendes sagen: Die Ein-/Ausgabe am Rechner ist irre kompliziert, selbst Doktoranden haben damit teilweise ihre Probleme. Daher besprechen wir dieses Thema auch ausführlich in Skriptteil VI. Da das Verständnis der Ein-/Ausgabe von so elementarer Bedeutung ist, geben wir in diesem Kapitel einen ersten Überblick. *„Und wie soll ich das verstehen, wenn's doch so schwer ist?“* Nun, wir geben unser Bestes und versuchen, erst mal das Problem ein bisschen einzukreisen, zu isolieren und dann zu verstehen. Und wie so oft: Dieses Unterfangen wird einfacher, wenn wir einen Blick in die Geschichte werfen.

41.1 The Good Old Days

Die Wurzeln der Computer und der Programmierung: Die Entwicklung und Grundprinzipien moderner Computer gehen auf den deutschen Ingenieur Konrad Zuse zurück. Während des zweiten Weltkrieges entwickelte Zuse zwei Rechner, die als Z2 und Z3 bekannt wurden. Die wesentliche Aufgabe dieser Maschinen spiegelt sich bereits im Wort „Rechner“ wieder: Es ging vornehmlich um die Erledigung komplexer Berechnungen, um die Ingenieure davon zu entlasten, die in der damaligen Zeit mit Rechenschiebern und Logarithmen-Tafeln arbeiteten. Die wesentliche Leistung von Konrad Zuse war die Entwicklung einer Maschine, die frei programmierbar war. Dadurch konnte ein und dieselbe Maschine für unterschiedliche Aufgabenstellungen verwendet werden, ohne dass Änderungen an der Hardware notwendig waren; alle notwendigen Anpassungen konnten durch Änderungen in der Software erreicht werden. Aufgrund dieser Flexibilität konnten diese Rechenmaschinen in vielen Ingenieursdisziplinen eingesetzt werden. Zu den potentiellen Einsatzgebieten gehörten die Berechnung ballistischer Flugkurven, die Entwicklung neuer

Optiken und die Berechung von Baustatiken.

Die gute alte Zeit der Lochkarte: Die nächste wichtige Frage betrifft die Art und Weise, wie Programme dargestellt wurden. Bis etwa zur Mitte der 70er Jahre waren Terminals eine Rarität und standen nur wenigen Mitarbeitern zur Verfügung. Vielfach wurden die Programme mittels großer Maschinen auf Lochkarten gestanzt. So eine Lochkarte hat eine ungefähre Größe von $18{,}7 \times 8{,}3\,\text{cm}$ und kann bis zu 80 Zeichen speichern[1]. Ein Problem war nun, dass die Erstellung einer Lochkarte etwa 30 Sekunden bis zu einer Minute gedauert hat[2]. Um unter diesen Randbedingungen dennoch den zeitlichen Aufwand einigermaßen in Grenzen zu halten, wurden die Daten nicht direkt in die Programme integriert, sondern strikt von ihnen getrennt. Diese strikte Trennung von Programm und (Eingabe-) Daten hatte sich sehr bewährt, denn so konnte man die Eingabedaten, die sich ebenfalls auf Lochkarten befanden, in einfacher Weise mit bereits getesteten Programm immer wieder neu kombinieren, ohne lange Testzyklen einlegen zu müssen. Das heißt, ein Lochkartenstapel bestand immer aus zwei *strikt* getrennten Teilen, wie die folgende Abbildung zeigt:

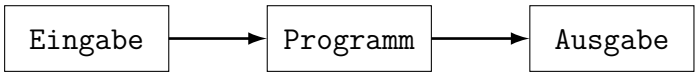


Der Stapelbetrieb (batch jobs): *„Was wurde nun mit diesen Lochkarten gemacht? Zum Essen waren sie doch sicherlich nicht gedacht. Gab es denn so einen breiten Schlitz im Rechner?“* Nein, so einen breiten Schlitz gab es nun wirklich nicht. Aber es gab Lochkartenleser. Man musste die Lochkarten in ein Fach legen. Dann kam der Operateur, hat die Lochkarten genommen, alles in einen einzigen Stapel gepackt, in den Lochkartenleser gelegt, eine Taste gedrückt, damit der Lochkartenleser die Lochkarten liest, und dann an der Konsole den Job zur Abarbeitung freigegeben. Durch diesen Ablauf wurde auch der Begriff Stapel- bzw. *batch*-Verarbeitung geprägt, der heute immer noch im Zusammenhang mit Shell-Scripts verwendet wird. Während der Abarbeitung kam noch eine dritte Komponente hinzu, die Ausgabe, die für gewöhnlich auf den Drucker geschickt wurde. Die Ausgabe bestand aus mindestens zwei Teilen, der eigentlichen Programmausgabe sowie einem Protokoll über den Ablauf des batch jobs. Mit anderen Worten: Bezüglich der Ein-/Ausgabe müssen wir drei Dinge voneinander unterscheiden: die Lochkarten für das Programm, die Lochkarten für die Eingabedaten und die Druckerausgabe. Zusamengefasst haben wir folgenden Ablauf:

[1] Das Format der Lochkarte ist übrigens der Hauptgrund dafür, dass die meisten Terminals und Kommandoeingaben ebenfalls eine Breite von 80 Zeichen haben.

[2] Ja, euer Informatikprofessor kennt das alles aus eigener Erfahrung, obwohl er erst Ende der 70er Jahre mit dem Studieren begonnen hat.

Ablauf beim Stapelbetrieb



Wie man deutlich sehen kann, sind zwar alle drei Teile miteinander verbunden, aber doch voneinander getrennt; insbesondere ist die Eingabe von der Ausgabe deutlich getrennt. Die Abfolge ist gemeinhin auch als „EVA“ (Eingabe, Verarbeitung, Ausgabe) Prinzip bekannt.

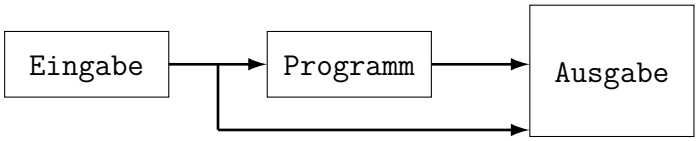
Wichtig?! *„Und warum ist das alles wichtig? Lerne ich hier noch 'was vernünftiges?“* Klar ist das wichtig, würden wir es sonst erzählen? Der Punkt ist, dass sich die mit dem Stapelbetrieb verbundene Form der Ein-/Ausgabe auch im Design der Ein-/Ausgabefunktionen in den Programmiersprachen niedergeschlagen hat. Ein Kernpunkt der Eingabe beim Stapelbetrieb ist, dass alle Eingabedaten bereits *vor* dem Programmstart fertiggestellt sein mussten; nachträglich konnte man keine Lochkarten mehr in den Stapel einfügen, denn sie wurden ja schon vor dem Programmstart eingelesen. Insofern ging es nur darum, die bereits eingelesenen Daten in das Programm zu bekommen. Anderweitige Dinge wie interaktive Abfragen oder gar interaktive Eingabeaufforderungen wurden nicht benötigt, denn *alle* Daten befanden sich ja bereits *vor* dem Programmstart tief unten im Rechner.

41.2 Interaktiver Terminalbetrieb

Die evolutionäre Weiterentwicklung: Mit der Zeit kamen Terminals auf, sodass man sein Programm bequem über Tastatur und Bildschirm entwickeln konnte. Aber dennoch war aufgrund des Programmiersprachen-Designs der Stapelbetrieb vorherrschend. Aber natürlich entstand der Wunsch, den Programmablauf interaktiv, also während der Programmbearbeitung, durch den Benutzer zu steuern. Das ging meist auch irgendwie, war aber immer eher ein krampfhafter *work around*; die damals vorherrschenden Programmiersprachen wie Fortran, Algol-60 und Pascal waren dafür einfach nicht ausgelegt. In diesem Punkt stellte die Programmiersprache C durch ihr *Design* eine echte Verbesserung dar!

Der interaktive Betrieb: Der Wunsch nach interaktiven Eingaben hat den oben dargestellten Betriebsablauf in einem Punkt wesentlich verändert: Zwar gibt es immer noch eine Eingabe, meist die Tastatur, das eigentliche Programm und eine gesonderte Ausgabe. *Aber (!)*, die Ausgabe stellt nicht nur die eigentliche Programmausgabe dar sondern *zusätzlich auch die Dateneingabe*:

Ablauf beim interaktiven Betrieb



Wo ist das Problem? *„Wo soll denn nun das Problem liegen? Ist doch schön, auf dem*

Bildschirm sehe ich auch gleich die Eingabe. Was soll denn da schief gehen?“ Also, für uns Profis ist dies kein Problem, für Programmieranfänger sehr schnell ein sehr großes. *„Ich habe beim Programmieren keine Probleme, ich löse welche ;-)“* Ja, ja, schön wär's. Im Übungsbetrieb sehen wir immer sehr viele Probleme. Das Hauptproblem ist, dass Programmieranfänger nicht mehr richtig zwischen der eigentlichen (Programm-) Ausgabe und der zusätzlich dargestellten Tastatureingabe unterscheiden können; für sie erscheint beides als eines. Aber leider ist es eben ganz anders. Durch das zusätzliche Darstellen der Eingabe wird das Erscheinungsbild der Ausgabe verändert, und durch das Darstellen der Ausgabe ist manchmal nicht mehr klar, was eigentlich die Eingabe war. Darauf gehen wir weiter unten nochmals etwas genauer ein.

Das zweite Problem: die intelligente Eingabe: Durch die Fortschritte im Bereich der angebotenen Datentypen und dem Wunsch nach möglichst viel Komfort haben die C-Designer intelligente Formatierungshilfen für die Eingabe entwickelt. Beispiele hierfür sind: `"%c"`, `"%d"` und `"%lf"`, die normalerweise der Eingabefunktion `scanf()` als Parameter übergeben werden. *„Also, ich könnte ja auf `"%d"` und `"%lf"` verzichten, mir würde `"%c"` reichen; den Rest berechne ich selbst.“* Da sagen wir nur: „Alter Angeber“. Diese Konvertierungen sind echt aufwändig und für die meisten Programmieranfänger erst mal viel zu schwer, um sie selbst zu entwickeln. Insofern können wir alle über diese intelligenten Formatierungen froh sein. Aber leider sind diese Formatierungshilfen die Ursache für manch „komisches“ Programmverhalten sowie diverse Endlosschleifen.

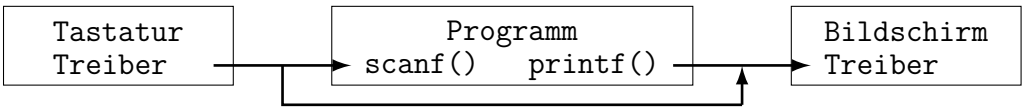
41.3 Die Funktionsweise der Ein- und Ausgabe: ein erster Einblick

Die Ausgabe: Die Ausgabe zu verstehen ist relativ einfach: Die von unserem C-Programm mittels `printf()` produzierten Ausgaben werden an das Betriebsystem geben, das seinerseits die Zeichen irgendwie auf den Bildschirm bringt. Auch wenn dieser Prozess hochkompliziert ist, so kann man ihn dennoch konzeptuell sehr leicht verstehen: Beginnend mit der aktuellen Position der Schreibmarke (dem Cursor) werden die Ausgaben dargestellt, die ihrerseits die Schreibmarke weiter nach rechts bzw. an den Anfang der nächsten Zeile verschieben. Darum brauchen wir uns eigentlich nicht weiter kümmern, aber Abschnitt 41.7 präsentiert hierzu noch ein paar kleine Beispiele.

Die Tastatur: Hier wird es schon richtig schwierig. Die Eingabe erfolgt in mindestens zwei Abschnitten. Zuerst werden alle Tastatureingaben lokal im Treiber (der Teil des Betriebssystems ist) zwischengespeichert und bearbeitet. *„Hä? Ich denke, die Eingabe geht an mein `scanf()`.“* Weit, sehr weit gefehlt. Dein `scanf()` merkt erst einmal gar nichts davon. Die Eingaben werden nicht einmal an das Programm übergeben. *„Was soll denn dieser unnötige Umstand?“* Das hat schon seinen Sinn und Zweck! Durch das Zwischenspeichern kann man seine Eingabe lokal mittels der `Del`- oder `Backspace`-Taste editieren bis man fertig ist. Ferner ist der Treiber so clever, dass er alle eingegebenen Zeichen von sich aus

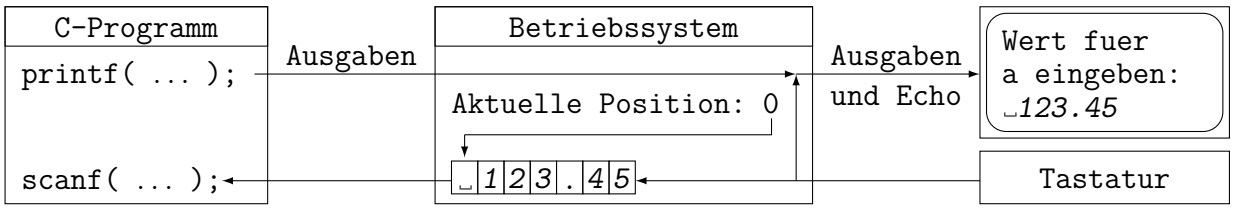
direkt auf dem Bildschirm darstellt (auch `Echo` genannt). Erst durch die Eingabe des Zeilenumbruchs `\n` betrachtet der Treiber die Zeile als fertig, erzeugt einen Zeilenumbruch auf dem Bildschirm und übergibt die fertige Eingabezeile an unser C-Programm. Unser obiges Modell der interaktiven Ein- und Ausgabe können wir wie folgt verfeinern:

Ablauf beim interaktiven Betrieb



Um das eben Gesagte nochmals zusammenzufassen: Alle Tastatureingaben werden erst einmal im Tastaturtreiber lokal gespeichert, hin und her editiert, gleichzeitig in Form eines Echos auf dem Bildschirm dargestellt, damit *wir* wissen, was wir eingegeben haben, und erst am Ende durch Eingabe des Zeilenumbruchs `\n` an das `scanf()` unseres Programms übergeben[3].

Aktuelle Lese- und Schreibmarke: Damit unsere Lese- (`scanf()`) und Schreibanweisungen (`printf()`) den Überblick behalten können, verwaltet das Betriebssystem sowohl eine aktuelle Leseposition (Lesemarke) als auch eine aktuelle Schreibeposition (Schreibmarke). *Alle* Ausgaben, also sowohl unsere eigenen Programmausgaben als auch das Echo unserer Tastatureingaben, werden immer relativ zu dieser Schreibmarke auf den Bildschirm gebracht. Jedes einzelne Zeichen bringt diese Schreibmarke um eine Position nach rechts. Sollte das Ende der Zeile erreicht sein oder wir einen Zeilenwechsel `'\n'` ausgeben, springt die Schreibmarke automatisch an den Anfang der nächsten Zeile. In ähnlicher Weise wird vom Betriebssystem eine Lesemarke verwaltet. Alle Programmeingaben (`scanf()`-Aufrufe) beziehen sich auf diese (virtuelle) Lesemarke. Nachdem nun die Eingabezeile fertig ist, wird sie an unser Programm übergeben. Die Eingabeposition wird bei jeder einzelnen Leseoperation weitergeschaltet, und zwar um eine Position je eingelesenem Zeichen. Diese Eingabeposition bezieht sich aber immer auf diejenige Zeile, die vom Tastaturtreiber bearbeitet wird und *nicht* auf das, was wir auf dem Bildschirm sehen, klingt trivial aber ist ganz wichtig. All dies ist in folgendem Bild zusammengefasst:



[3]Wie so oft ist selbst diese Darstellung *sehr* stark vereinfacht. Aber dennoch spiegelt sie die Wirklichkeit recht gut wider.

Obiges Bild zeigt die folgenden Dinge:

1. Das Programm hat die Zeichenfolge `Wert fuer a eingeben:` mittels des Betriebssystems auf dem Bildschirm ausgegeben.
2. Der Nutzer hat die Zeichenfolge `␣123.45` über die Tastatur eingegeben. Diese Zeichenfolge wird vom Betriebssystem in einem internen Register abgelegt. Die aktuelle Leseposition ist `0`, da noch keines der Zeichen von unserem C-Programm verarbeitet wurde.
3. Die eingetippte Zeichenfolge wurde vom Betriebssystem in Form eines Echos auf dem Bildschirm ausgegeben.
4. Zur Veranschaulichung stellen wir die Eingaben mittels leicht geneigter Zeichen dar.

Das weitere Verhalten des Gesamtsystems hängt nun davon ab, welche Formatierung dem `scanf()` übergeben wurde. Je nach Formatierung versucht `scanf()` mehr oder weniger Zeichen auf einmal zu verarbeiten.

Wichtiger Hinweis: Niemand sollte jemals, in keiner noch so abgefahrenen Situation auf den Gedanken kommen, dass diese Eingabemarke ein Pointer ist (siehe die folgenden Kapitel), den man irgendwie direkt verändern kann. Sowohl die Ein- als auch die Ausgabemarke werden vom Betriebssystem verwaltet und sind für uns nicht direkt zugreifbar.

41.4 `scanf()` und seine intelligenten Formatierungen

Die weitere Verarbeitung der Eingabe aus obigem Beispiel hängt jetzt von den Formatierungsanweisungen ab. Wir können einfach mal wie folgt unterscheiden:

1. `"%c"`:
   Bei der Formatierung `"%c"` wird das nächste Zeichen gelesen, der angegebenen Variablen übergeben und die Eingabemarke um eine Position nach rechts geschoben. Das wiederholte Ausführen der Anweisung `scanf( "%c", & c )` hätte für die Variable `c` folgenden Effekt:

Arbeitsschritt:	1	2	3	4	5	6	7	8
c:	’ ’	’1’	’2’	’3’	’.’	’4’	’5’	’\n’

   Wie illustriert, wandert die Eingabemarke bei jedem `scanf()` um eine Position weiter. Simultan dazu nimmt die übergebene Variable `c` als Wert immer das nächste Zeichen an.

2. `"%d"`:
   Bei ganzzahligen Argumenten verhält sich die Funktion `scanf()` *komplett* anders! Die Funktionsweise von `"%d"` könnten wir wie folgt darstellen:

   ```
   1 int scanf_int()
   2     {
   ```

```
3         char c;
4         int  i;
5         do scanf( "%c", & c );
6         while( c == ' ' || c == '\t' || c == '\n' );
7         i = 0;
8         while( c >= '0' && c <= '9' )
9         {
10           i = i * 10 + c - '0';
11           scanf( "%c", & c );
12        }
13        ungetc( c, stdin );
14        return i;
15    }
```

Im Folgenden gehen wir die Funktion `scanf_int()` Schritt für Schritt durch.

Zeilen 5 und 6:
Hier überliest die Funktion `scanf_int()` alle Leerzeichen, Tabulatoren und Zeilenumbrüche und bleibt erst dann stehen, wenn ein Nicht-Leerzeichen (Whitespace) in der Eingabe erscheint.

Zeile 7:
Hier wird unsere `int`-Variable `i` mit dem Wert `0` initialisiert.

Zeilen 8 bis 12:
In diesen fünf Zeilen werden solange Zeichen eingelesen, wie es sich um Dezimalziffern handelt. Jede dieser Ziffer wird in Zeile 10 dazu verwendet, den Wert der Variablen `i` zu aktualisieren. Die Schleife bricht ab, wenn ein Zeichen kommt, das kein Dezimalzeichen ist (Zeile 8).

Bezogen auf unser obiges Beispiel nehmen Eingabemarke und Variablen nacheinander folgende Werte an.

Eingabemarke:	0	1	2	3	4
`c`:	`' '`	`'1'`	`'2'`	`'3'`	`'.'`
`i`:	0	1	12	123	123
Neue Eingabemarke:	1	2	3	4	5

Durch das letzte Lesen (des Zeichens `'.'`) ist die Funktion `scanf_int()` um eine Position zu weit gegangen, was im nächsten Schritt korrigiert wird.

Zeile 13:
Durch die Anweisung `ungetc()` wird das angegebene Zeichen (der Punkt `'.'` in unserem obigen Beispiel) wieder in die Eingabe zurückgeschrieben. Durch diese Anweisung wird auch die aktuelle Eingabemarke auf die Position `4` zurückgesetzt (am Ende von Zeile 12 hatte diese bereits den Wert `5`).

Zeile 14:
Diese Zeile gibt das Ergebis (123 in unserem Beispiel) zurück und beendet die Funktion `scanf_int()`.

3. `"%lf"`:
   Die Verarbeitung der Eingabe erfolgt hier ähnlich wie im vorherigen Fall. Nach dem Einlesen der „Zahl“ `123` würde der Punkt überlesen werden und anschließend die „Zahl“ `45` gelesen. In einem abschließenden Schritt würde aus beiden Werten die endgültige Zahl `123.45` gebildet und zurückgegeben werden.

Diese zusätzliche Intelligenz der formatierten Eingabe kann nun zu Problemen führen, wie im nächsten Abschnitt gezeigt wird.

41.5 Endlosschleife durch formatierte Eingabe

In diesem Abschnitt nehmen wir wieder Bezug auf das Beispiel, das wir bereits in Abschnitt 41.3 (Seite 161) eingeführt haben. In Abschnitt 41.4 Punkt 2 (Seite 162) haben wir dargestellt, dass nach einem ersten Aufruf von `scanf_int()` der Wert 123 zurückgegeben wird und die Eingabemarke auf Position `4` (zurück-) gesetzt wird.

Was passiert nun, wenn wir bei der gegebenen Eingabe nochmals einen ganzzahligen Wert lesen wollen, also `scanf( "%d", & i )` (bzw. `scanf_int()`) erneut aufrufen? Schauen wir einfach in die Implementierung. In den Zeilen 5 und 6 werden wieder die führenden Leerzeichen überlesen. Da gleich der Punkt `'.'` kommt, passiert nichts weiter. Anschließend werden in den Zeilen 8 bis 12 alle Dezimalziffern eingelesen und für die Wertberechnung verwendet. Da aber bereits der Punkt `'.'` eingelesen wurde, passiert auch hier nichts. Abschließend wird in Zeile 13 der Punkt `'.'` wieder in die Eingabe zurück geschrieben und die Eingabemarke erneut auf die Positon `4` zurückgesetzt. Durch diesen Mechanismus würde der Punkt bei der nächsten Eingabe abermals gelesen werden. Sollte wieder eine Zahl gelesen werden, wiederholt sich der gesamte Prozess aufs neue.

Wie wir jetzt sehen, ist die Eingabe um keine einzige Position vorangeschritten, obwohl wir die Funktion `scanf_int()` erneut aufgerufen haben. Und egal wie oft wir jetzt die Funktion `scanf_int()` aufrufen, passiert nichts, was in vielen Fällen zu einer Endlosschleife führt.

Natürlich entsteht die Frage, ob man nicht irgendwie einen Ausweg aus dieser unerwünschten Endlosschleife findet. Eine Möglichkeit sieht wie folgt aus: Die Funktion `scanf()` teilt uns über ihren Rückgabewert mit, wie viele Argumente sie erfolgreich konvertiert hat. In unserem letzten Fall, in dem wir leider den Punkt gelesen haben, wäre das `0`, da der Punkt eine Konvertierung der Eingabe verhindet. Entsprechend könnten wir in so einem Falle alle Zeichen bis zum Zeilenwechsel überlesen, wie folgender Programmausschnitt für die erste Variable zeigt:

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

```
 3 int main( int argc, char **argv )
 4     {
 5        char c;
 6        int a, b, F;
 7        printf( "Bitte Seite a eingeben: " );
 8        while ( scanf( "%d", & a ) == 0 )
 9        {
10          do scanf( "%c", & c );
11          while( c != '\n' );
12          printf( "Fehlerhafte Eingabe, bitte wiederholen\n" );
13        }
14        // ... hier folgt der Rest des Programms ...
15     }
```

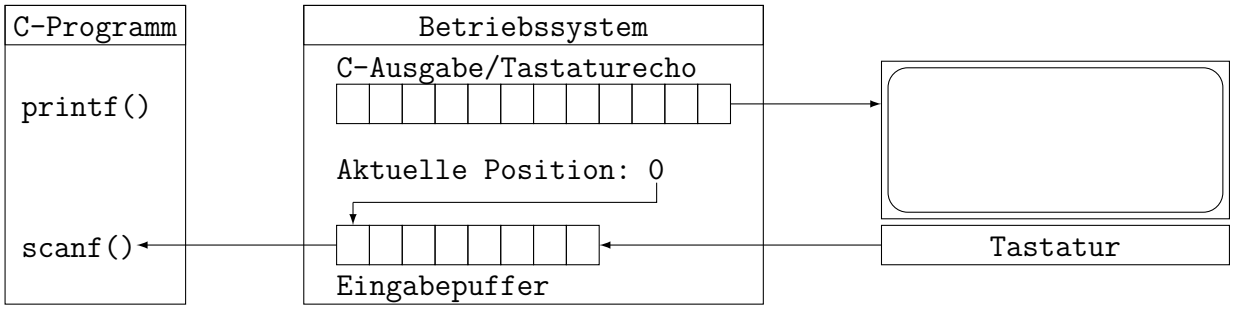
Dieser Ansatz funktioniert wie gewünscht, da `scanf( "%c", & c )` einfach das nächste Zeichen liest, unabhängig davon, um was für ein Zeichen es sich handelt. Durch die Schleife in den Zeilen 8 bis 13 wird der Vorgang solange wiederholt, wie die Eingabe fehlerhaft ist.

41.6 „Übersprungene“ Eingabe

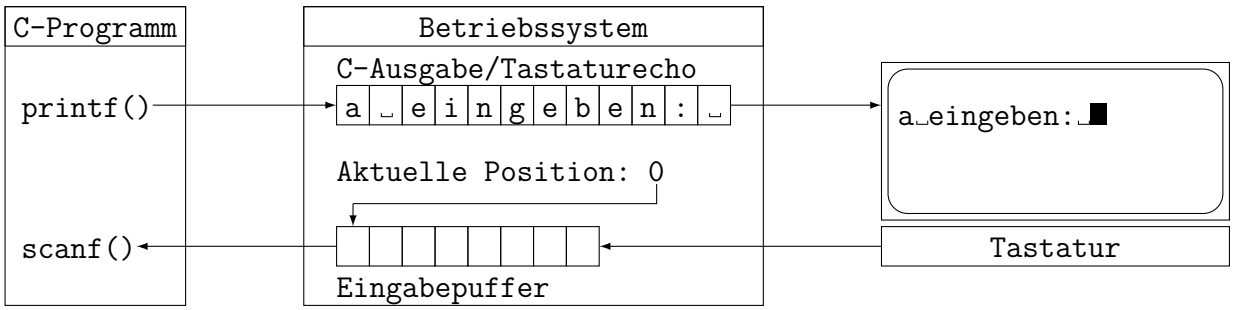
Manchmal haben Programmieranfänger den Eindruck, dass Teile ihrer Eingaben einfach übersprungen werden. Dies ist natürlich in der Regel nicht der Fall, aber der Eindruck ist tatsächlich vorhanden und sogar plausibel. Hierfür schauen wir uns einfach nochmals unser erstes Programm an, die Berechnung der Fläche eines Rechtecks, wobei wir nur die ersten beiden Ein- und Ausgabeanweisungen dargestellt haben. Ferner haben wir aus Platzgründen die Ausgabetexte auf das Notwendigste verkürzt:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5        int a, b, F;
 6        printf( "a eingeben: " );
 7        scanf( "%d", & a );
 8        printf( "b eingeben: " );
 9        scanf( "%d", & b );
10     }
```

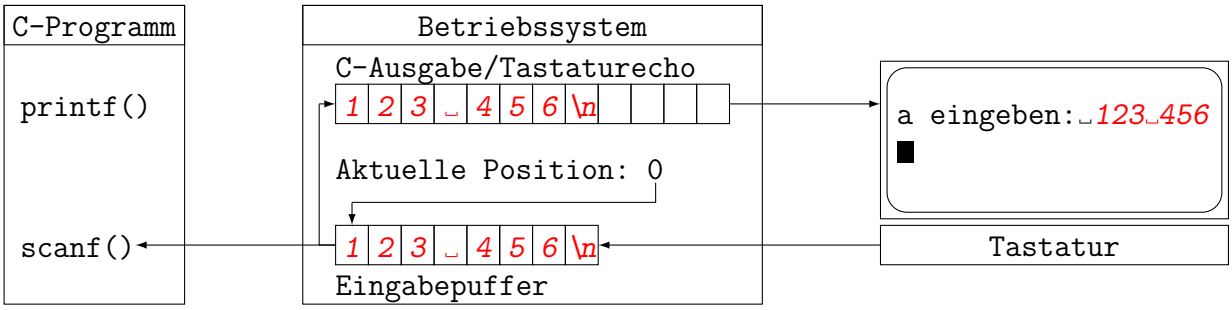
In Anlehnung an das weiter oben wiedergegebene Bildchen zeigen die folgenden ein stilisiertes C-Programm, einen kleinen Ausschnitt aus dem Betriebssystem sowie ein Terminal einschließlich Tastatur. Zur besseren Darstellung sind die Eingaben wieder in *geneigter*, roter Schrift und die ausgegebenden Leerzeichen durch das Zeichen ␣ dargestellt. Bis Zeile 5 wurden noch keine Ein-/Ausgaben getätigt, sodass die Situation wie folgt aussieht:



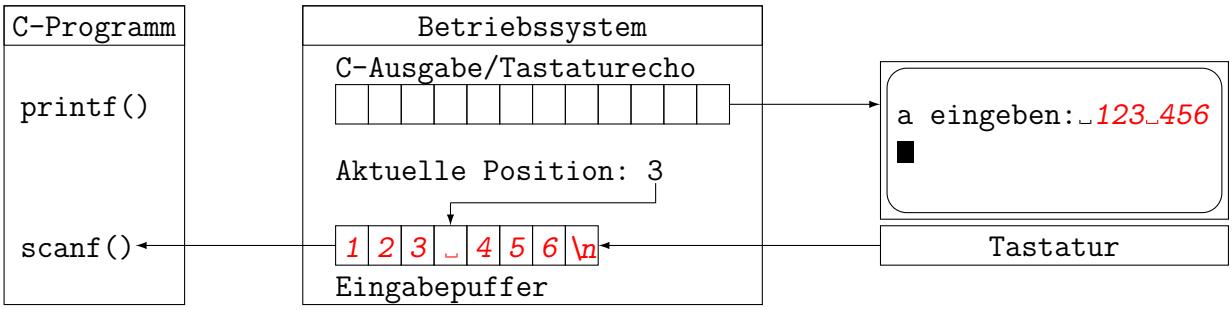
Nun wird in Zeile 6 der Text `a␣eingeben:␣` ausgegeben. Dieser Text wird vom Betriebssystem in einem Register zwischengespeichert und ab der aktuellen Position der Schreibmarke, auch als Cursor bezeichnet, auf dem Bildschirm ausgegeben. Nachdem der Text auf dem Bildschirm angekommen ist, wird das Ausgaberegister wieder vom Betriebssystem geleert. Nach Beendigung der Programmzeile 6 haben wir also folgendes Bild:



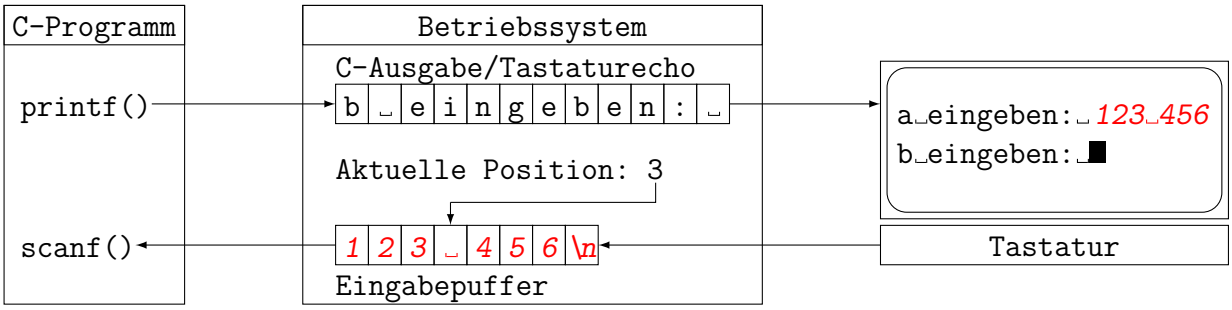
Bisher haben wir noch nichts eingegeben. Entsprechend ist der vom Betriebssystem verwaltete Eingabepuffer leer. Nun aber will unser Programm in Zeile 7 etwas einlesen. Hierzu führt es die `scanf()`-Anweisung entsprechend aus. Da aber der Eingabepuffer noch leer ist, muss unser Programm warten. Wenn wir nun etwas über die Tastatur eingeben, so gelangen diese in den Eingabepuffer des Betriebssystems. Dort werden diese Eingaben wie in Abschnitt 41.3 erläutert, festgehalten und lokal editiert. Ferner werden alle Eingaben mittels des Ausgaberegisters auf dem Bildschirm (aber der Schreibmarke) dargestellt, damit wir sehen, was wir tippen. Erst durch Eingabe des Zeilenwechsel `'\n'` wird dieser Vorgang beendet und die Eingabe für unser C-Programm zum Verarbeiten „freigegeben“. Wichtig für das Verständnis ist außerdem, dass die Schreibmarke durch die Eingabe des Zeilenwechsels und das folgende Echo auf den Bildschirm auf den Anfang der nächsten Zeile wechselt. Im folgenden Beispiel haben wir zwei Zahlen auf einmal eingegeben, die beide in den Eingabepuffer gelangen und auf dem Bildschirm abgebildet werden. Zur Erinnerung: das lokale Editieren wird erst durch die Eingabe des Zeilenwechsels `'\n'` beendet und nicht schon dann, wenn das `scanf()` des C-Programms zufrieden sein könnte. Wir haben nun folgende Situation:



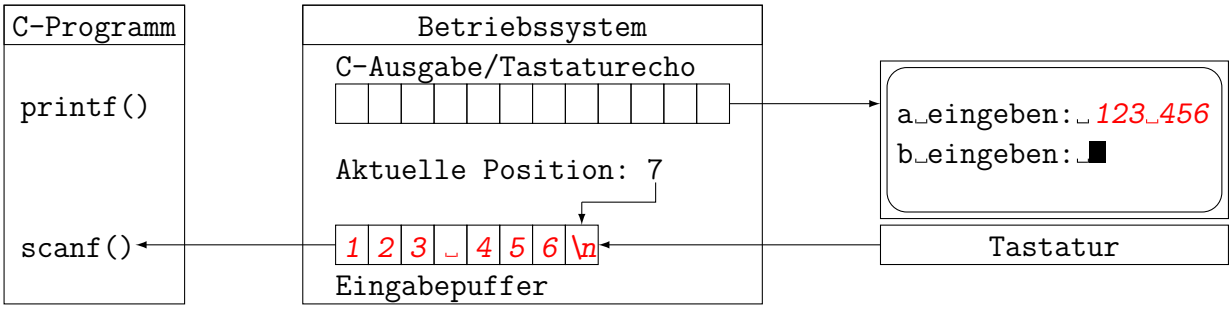
Wie eben schon gesagt, ist die aktuelle Leseposition `0` und der Eingabepuffer mit zwei Zahlen vom Typ Integer gefüllt. Nun will das `scanf()` eine Ganzzahl `"%d"` lesen. Dazu gehören die ggf. vorangestellten Leerzeichen und die drei Ziffern `'1'`, `'2'` und `'3'`. Das nachfolgende Leerzeichen wird zwar eingelesen, woraufhin die Einleseschleife der Funktion `scanf()` abbricht, aber anschließend wieder automatisch in die Eingabe zurückgeschrieben, da ja das Leerzeichen nicht zur Zahl gehört. Dies bedeutet, dass die `scanf( "%d", & a )`-Anweisung genau eine Zahl nebst der möglicherweise vorangestellten Leerzeichen, Tabulatoren und Zeilenwechsel „verdaut“ hat. Der Zustand unmittelbar am Ende von Zeile 7 ist in folgendem Bild illustriert:



Wie zu sehen ist, verbleibt die zweite Zahl `456` sowie der Zeilenwechsel weiterhin im Eingabepuffer des Betriebssystems und die aktuelle Leseposition verbleibt an der Stelle `3`. Durch Zeile `8` wird nun die nächste Eingabeaufforderung `"b eingeben: "` ausgegeben. Dadurch wird die Ausgabemarke hinter das Leerzeichen gesetzt, das dem Doppelpunkt folgt. Dies klingt sicherlich recht plausibel und einleuchtend. Obwohl nun durch die eben erwähnte Ausgabe die Schreibmarke versetzt wurde, bleibt die Position der Eingabemarke unverändert: sie steht immer noch direkt hinter der Zahl 123, die sich eine Zeile höher befindet. Diese Situation ist in folgender Grafik nochmals dargestellt:



Nun kommt Zeile 9 zur Wirkung. Das `scanf()`-Anweisung will wieder eine Zahl vom Typ Integer lesen und fragt das Betriebssystem nach weiteren Zeichen. Da sich nun aber noch welche im Eingabepuffer befinden, wird unser C-Programm nicht verzögert sondern direkt bedient. Es bekommt die Zeichen `456` sowie den Zeilenwechsel, den es aber wieder zurückschreibt, da es sich nicht um eine Zahl handelt. Mit anderen Worten: Unser C-Programm erhält die nächsten Zeichen und kann diese als Zahl mit dem Wert `456` der Variablen `b` zuweisen. Obwohl jetzt die zweite Zahl eingelesen wurde, sehen wir auf dem Bildschirm keine Veränderung. Dieses Verhalten bedarf einiger Gewöhnung und ist für die meisten (Programmieranfänger) etwas kontraintuitiv. Die folgende Grafik fasst diese Situation nochmals zusammen:



Um das Gesagte noch einmal zusammenzufassen: Das Betriebssystem verwaltet für uns zwei Marken, eine Lese- und eine Schreibmarke. Ferner verwaltet das Betriebssystem einen Eingabepuffer, in dem die vom Nutzer getätigten Eingaben zwischengespeichert und vom Programm *bei Bedarf* (durch `scanf()`-Anweisungen) ausgelesen werden. Auf diese beiden Marken haben wir keinen direkten Einfluss außer durch entsprechende Ein- und Ausgabeanweisungen. Da das Betriebssystem gezwungen ist, sowohl die Ein- als auch die Ausgaben auf dem selben Bildschirm darzustellen, sind diese beiden Bereiche schwer voneinander zu unterscheiden, sodass manchmal die Orientierung etwas schwerer fällt. Abschließend sei erwähnt, dass andere Programmiersprachen ihre Ein-/Ausgabe anders organisieren.

41.7 Beispiele zur formatierten Ausgabe

Wie versprochen präsentieren wir hier noch ein paar kleine Beispiele. Die C-Ausgabe ist relativ leicht zu verstehen, wenn man die folgenden drei Regeln im Kopf behält:

Regel 1: Die Ausgabe erfolgt *immer* an der aktuellen Position der Schreibmarke.

Regel 2: Es macht keinen Unterschied, ob man alle Ausgaben in einen `printf()`-Aufruf packt, oder diese über mehrere Aufrufe verteilt.

Regel 3: Die Schreibmarke wechselt auf den Anfang der nächsten Zeile, wenn das Ende der Zeile erreicht ist, ein `'\n'` ausgegeben oder etwas eingegeben wurde.

Die Beispiele der folgenden Tabelle konzentrieren sich vor allem auf Regel 2. In allen Anweisungen gilt: `int i = 123; char c = 'A';`

Bildschirmausgabe	Anweisungen und Alternativen
abc def ▮	1. `printf( "abc def" );` 2. `printf( "abc " ); printf( "def" );`
abc def ▮	1. `printf( "abc def\n" );` 2. `printf( "abc def" ); printf( "\n" );` 3. `printf( "abc de" ); printf( "f\n" );`
i= 123▮	1. `printf( "i= %d", i );` 2. `printf( "i= " ); printf( "%d", i );`
i= 123 ▮	1. `printf( "i= %4d\n", i );` 2. `printf( "i= ", ); printf( "%4d\n", i );`

alles so sinnlos
 	
▮
	1. `printf( "alles ist sinnlos\n\n");`
	2. `printf( "alles ist sinnlos\n");`
	`printf( "\n");`
	3. `printf( "alles ist sinnlos");`
	`printf( "\n"); printf( "\n");`

 fun fun fun▮	1. `printf( "\nfun fun fun");` 2. `printf( "\n"); printf( "fun fun fun");` 3. `printf( "\n");` `printf( "fun fu"); printf( "n fun");`
 hier ist 123ABC▮	1. `printf( "hier ist %d%c%c%c", i, c, c+1, c+2 );` 2. `printf( "hier ist %d", i );` `printf( "%c%c%c", c, c + 1, c + 2 );`

Teil V

Intermediate C: The Juicy Stuff

Kapitel 42

Inhalte dieses Skriptteils

Bisher haben wir die grundlegende Herangehensweise an das Programmieren besprochen, die einfachen Sprachkonzepte kennengelernt und darüber geredet, wie alles von der Hardware bearbeitet wird und welche Rolle der Compiler dabei spielt. Nun kommen endlich die Dinge, die das Programmieren interesssant machen. Diese sind vor allem Erweiterungen zu den Themen Ausdrücke, Arrays, Funktionen und Strukturen. Mit dem bisherigen Vorwissen sollten diese Sprachkonstrukte keine besonderen Hürden darstellen. Um wirklich zu verstehen, was vor sich geht, werden wir immer wieder den Bezug zur Hardware suchen und auch finden. Im Weiteren werden wir die folgenden Themen der Reihe nach behandeln:

Kapitel	Inhalt
43	Ausdücke: Teil II
44	Programmierung eigener Funktionen
45	Zeiger und Adressen
46	Arrays und Zeiger-Arithmetik
47	Funktionen mit Arrays und Zeigern
48	Rekursion
49	Mehrdimensionale Arrays
50	Zeichenketten bzw. Datentyp *String*
51	Kommandozeile: `argc` und `argv`
52	Programmabstürze und sicheres Programmieren
53	Zusammengesetzte Datentypen: `struct`
54	Selbstdefinierte Datentypen
55	„Module“ und getrenntes Übersetzen
56	Variablen: Sichtbarkeit und Lebensdauer
57	`void`: der besondere Datentyp

Leider wird nicht alles einfach zu verstehen sein. Eines dieser Themen sind Zeiger. Diese verlangen vom Lernenden etwas Geduld! Wir als Lehrpersonal haben sie sowie, zumindest solange wie Ihr Euch ernsthaft bemüht; nun müßt Ihr die Geduld nur noch mit Euch haben!

Auch in diesem Abschnitt gilt das schon öfter gesagte: Don't panic! Einfach mal ausprobieren und vor allem die Übungsaufgaben bearbeiten. Und bei Problemen helfen wir alle gerne weiter.

So, let's do it and have some (programming) fun!

Kapitel 43

Ausdrücke: Teil II

Ausdrücke hatten wir bereits in Kapitel 22. Aber das war nur ein erster Überblick, in dem wir Ausdrücke nach arithmetischen, logischen Vergleichs- sowie logischen Verknüpfungsoperatoren sortiert haben. Zusätzlich haben wir in Kapitel 37 über das Konzeptes des Datentyps und dessen Bedeutung gesprochen. In diesem Kapitel nun erweitern wir unsere Kenntnisse über Ausdrücke und lernen vor allem schöne Kurzformen. Aber es sei hier wieder daran erinnert: Die Programmiersprache C ist von Experten für Experten entwickelt worden, die alle wissen, was sie tun. Daher sei eindringlich darauf hingewiesen, dass man auch bei den Kurzformen sehr sorgfältig programmieren sollte.

43.1 Fachvokabular: Evaluation, Auswertungsreihenfolge, Präzedenz, Seiteneffekte

Da wir hier an einer Uni sind, müssen wir auch von Zeit zu Zeit ein wenig Fachvokabular lernen. Im Rahmen von Ausdrücken benötigen wir vier Fachbegriffe: Evaluation, Auswertungsreihenfolge, Präzedenz und Seiteneffekt. Diese Fachbegriffe klingen recht hochgestochen, aber so ist es nun mal. Unsere Empfehlung: Einfach locker bleiben!

Evaluation: Dieses Wort heisst so viel wie Auswerten, Bewerten, Beurteilen und bedeutet im Zusammenhang mit Programmieren nichts anderes als „Ausrechnen“. Dieses Ausrechnen bezieht sich nicht nur auf eine ganze Gleichung sondern vor allem auf jeden einzelnen Bestandteil. Ein kleines Beispiel:

```
1  i = 4;
2  j = 2*(1 + 3);
3  k = i + j + 2;
```

Das Ergebnis dieses komplexen Programms sollte jedem sofort klar sein. Die drei Variablen bekommen die Werte 4, 8 und 14. Evaluation heißt nun, dass auf den

rechten Seiten der Wert jedes einzelnen Terms bestimmt wird. In der ersten Zeile kommt dabei völlig verblüffend der Wert vier heraus. In der zweiten Zeile der Wert acht. Siehe aber auch noch den übernächsten Absatz.

Die dritte Zeile wird in ähnlicher Form abgearbeitet, was jedem klar sein sollte. Aber bereits hier sieht man, das Evaluation doch etwas mehr ist: Um den Wert von `i` bestimmen zu können, muss erst die Adresse der Variablen auf den Adressbus gelegt werden, damit sich die richtige Schublade öffnet, und anschließend muss der entsprechende Wert vom Speicher in eines der Zwischenablagen (Register) der CPU transportiert werden (siehe auch nochmals Kapitel 36). Das Gleiche wird für die Variable `j` und die Konstante 2 abgewickelt.

Zum Schluss noch zwei Bemerkungen: Da Compiler-Bauer auch nicht gerade doof sind, wird der Wert der rechten Seite der zweiten Zeile schon vom Compiler zur Übersetzungszeit ausgerechnet, sodass „quasi“ `j = 8;` im Maschinenprogramm steht. Zweitens wird auch die linke Seite ausgewertet, evaluiert. Das Ergebnis ist jedes Mal die Adresse der entsprechenden Variablen.

Auswertungsreihenfolge: Dies betrifft die *Reihenfolge*, in der die einzelnen Terme evaluiert werden und wird häufig mit Präzendenz gleichgesetzt und/oder verwechselt. Man denkt „normalerweise“, dass sich die Auswertungsreihenfolge an den Vorrangregeln orientiert. Entsprechend würden im Fall

```
1 i = j*(k + m);
```

die meisten folgende Reihenfolge annehmen: `k`, `m`, `j`, da aufgrund der Regel „Punkt-vor Strichrechnung,“ die Klammer zuerst ausgerechnet werden muss. Nun, das könnte man alles so vermuten. Ist aber nicht so! *„Wie, rechnet C etwa falsch?“* Nein, C rechnet schon richtig, nur darf sich C-Compiler selbst überlegen, in welcher Reihenfolge er die einzelnen Terme evaluiert. *„Ich verstehe nur Bahnhof . . .“*, werden jetzt einige sagen. Ja, das Komplizierte an der Sache ist, dass man die beiden Begriffe Evaluieren und Ausrechnen auseinander halten muss.

Ok, nochmals von vorne: beim Evaluieren wurden die richtigen Schubladen im Arbeitsspeicher geöffnet und deren Inhalte in die Zwischenablagen (Register) der CPU transferiert. Beim Ausrechnen geht es nun darum, diese Zwischenablagen gemäß der Rechenregeln mittels der ALU zu verknüpfen. Bei letzterem gibt es keine Freiheiten, bei ersterem hat der C-Compiler die freie Wahl. Obiger Ausdruck `i = j*(k + m)` muss man sich wie folgt vorstellen, wobei `tmp_x` Register der CPU symbolisieren:

```
1 tmp_j = j;               // auswerten j
2 tmp_k = k;               // auswerten k
3 tmp_m = m;               // auswerten m
4 tmp_x = tmp_k + tmp_m;   // berechnen k + m
5 tmp_i = tmp_x * tmp_j;   // berechnen j*(k + m)
6 i = tmp_i;
```

Und zur Wiederholung: Gemäß Sprachreport hat der Compiler die freie Wahl, in welcher Reihenfolge er die einzelnen Terme evaluiert (Zeile 1 bis 3 in obigem Beispiel), solange er am Ende alles richtig zusammenrechnet (Zeile 4 bis 6 in obigem Beispiel). Entsprechend wären aus Sicht der Sprache C folgende Variante *alle* richtig:

```
1 tmp_j=j; tmp_k=k; tmp_m=m;
2 i = (tmp_k + tmp_m)*tmp_j;
```

```
1 tmp_k=k; tmp_j=j; tmp_m=m;
2 i = (tmp_k + tmp_m)*tmp_j;
```

```
1 tmp_m=m; tmp_k=k; tmp_j=j;
2 i = (tmp_k + tmp_m)*tmp_j;
```

```
1 tmp_m=m; tmp_j=j; tmp_k=k;
2 i = (tmp_k + tmp_m)*tmp_j;
```

„Ist das eigentlich wichtig? Ist das nicht total egal?“ Nun, im Zusammenhang mit den nächsten Abschnitten ist dieser Sachverhalt sogar *sehr* wichtig. Hätten wir einen Ausdruck wie `d=sin(M_PI/2.0)*cos(0.0)` dann würde immer `d = 1` herauskommen, aber wir wüssten nicht, in welcher Reihenfolge die beiden Funktionen aufgerufen werden. Hätten diese beiden Funktionen jeweils eine Ausgabeanweisung (was sie zwar nicht haben, hier aber keine Rolle spielt, da es ein Beispiel ist), wüssten wir vorher nicht, in welcher Reihenfolge die Ausgaben erscheinen. Wir können uns also nicht darauf verlassen, dass die Funktion `sin()` vor der Funktion `cos()` aufgerufen wird.

Hinweis: Das Gesagte betrifft arithmetische Ausdücke; logische Ausdrücke werden von links nach recht abgearbeitet.

Präzedenz: Dies betrifft die Rangfolge der einzelnen Operatoren. Bei arithmetischen Ausdrücken ist dies einfach Punkt- vor Strichrechnung, Vergleiche und logische Verknüpfungen haben geringere Bindung (Präzedenz). Aber da man sich das eh nicht richtig merken kann, haben wir die Präzedenz-Regeln in den Anhang gepackt; auch fortgeschrittene Programmierer müssen dort des öfteren mal nachschauen.

Seiteneffekt: Ein Seiteneffekt liegt immer dann vor, wenn sich etwas verändert hat. Für uns heißt das erst einmal, dass sich der Wert einer Variablen verändert hat. Also `i = 4711` hat offensichtlich einen Seiteneffekt. Es gibt aber noch andere Ausdrücke (siehe unten), die auch Seiteneffekte haben und im Zusammenhang mit der Auswertungsreihenfolge wird es plötzlich recht kompliziert. Streng genommen liegen auch bei allen Ein- und Ausgaben Seiteneffekte vor, da sich etwas verändert hat.

43.2 Zuweisung

Nach dem etwas kleinlich und philosophisch anmutenden letzten Abschnitt machen wir jetzt mal etwas einfaches, konkretes, nämlich Zuweisungen. Ein Beispiel wie `i = 5;` haben wir zu genüge besprochen und mit hoher Wahrscheinlickeit von uns allen auch bis ins Detail (einschließlich Hardware) voll durchdrungen. Nun ist es so, dass eine Zuweisung wieder ein Ausdruck ist. *„Wie, ein Ausdruck?“* Ja, ein Ausdruck, als hätte man einfach nur `5` geschrieben.

„Irgendwie verstehe ich nicht, was das bedeuten soll: Eine Zuweisung ist auch ein Ausdruck.“ Don't panic. Auch vielen fortgeschrittenen Programmierern fällt es schwer, dies zu verstehen. Stellen wir uns vor, dass in obigem Beispiel die `5`, die von der CPU zum Arbeitsspeicher geschickt wurde, vorher in einen Briefumschlag gelegt wurde. Dann sorgt der Compiler dafür, dass eben eine zweite `5` in einen zweiten Briefumschlag gelegt wurde, der dann zurück ins (Haupt-) Programm geschickt wird, damit dort jemand machen kann, was auch immer er damit machen möchte.

„Und was, wenn niemand diesen Briefumschlag haben will?“ Macht nichts, dann landet er eben in Ablage-P. *„Wenn das stimmt, dann kann man einfach einen Ausdruck hinschreiben und diesen mit einem Semikolon abschließen ohne das Ergebnis irgendwo hinzuschieben . . .?* Richtig. Das C-Programm `int main( int argc, char **argv ){ 5+3; }` wäre völlig korrekt. Es wird der Wert acht ausgerechnet und dem Hauptprogramm übergeben, wo dieser Wert einfach verpufft.

„Ok ok, aber ist das nicht völlig sinnloser, philosophischer Dünnpfiff? Nein, ganz und gar nicht. Es könnte ja sein, dass einer der Terme, z.B. ein Funktionsaufruf, einen gewünschten Seiteneffekt hat. Beispielsweise ist jeder Aufruf von `printf()` ein derartiger Fall: Die Funktion `printf()` gibt immer einen Wert zurück, der in den wenigsten Fällen weiter verwendet wird. Einfach mal ausprobieren, was da so alles bei `printf()` zurückkommt, und auch mal die Dokumentation lesen ;-)

Ein zweiter Anwendungsfall sind ganz normale, geschachtelte Zuweisungen. Im Folgenden ein paar Beispiele, bei denen unbedingt die Präzedenz (was bindet stärker, siehe oben und Anhang) beachtet werden sollte:

```
1 i = (j = 5);        // i = 5; j = 5;
2 i = j = 5;          // i = 5; j = 5;
3 i = (j = 5) * 2;    // i =10; j = 5;
4 i = j = 5 * 2;      // i =10; j =10;'*' bindet staerker als '='
5 i = (j = 5) > 3;    // i = 1; j = 5;
6 i = j = 5 > 3;      // i = 1; j = 1;'>' bindet staerker als '='
```

Da Zuweisungen auch Ausdrücke sind, können diese Zuweisungen auch überall dort stehen, wo Ausdrücke erwartet werden, beispielsweise im Bedingungsteil einer `if`-Abfrage:

```
1 i = 3;                          1 i = 3;
2 if ( i = 5 )                    2 if ( i == 5 )
3       printf( "true\n" );       3       printf( "true\n" );
4 else printf( "false\n" );       4 else printf( "false\n" );
```

Beide Programme sehen auf den ersten Blick identisch aus. Sind sie aber nicht. Im linken Programm wird `true` ausgegeben, im rechten `false`. Warum, ist einfach erklärt: Im linken Beispiel steht im Bedingungsteil eine Zuweisung. Diese wird ausgeführt. Also erhält `i` den Wert `5`. Anschließend wird geschaut, ob diese Zuweisung einen Wert ungleich null hatte.

Da dies zutrifft, wird also `true` ausgegeben. Im rechten Teil wird einfach nur überprüft, ob die Variable `i` den Wert `5` hat. Da dies nicht der Fall ist, wird `false` ausgegeben. Klar?

43.3 Kurzform bei Zuweisungen

Sehr häufig hat man Variablenzuweisungen, in denen auf der rechten sowie der linken Seite der Gleichung die selbe Variable steht. Beispiele: `i = i + 4` und `d = d * 5`. Für diese häufigen Fälle gibt es die Kurzform: Operator gefolgt vom Gleichheitszeichen. Damit's klar wird, schauen wir doch am besten auf folgende Beispiele:

Normalform:

```
1 i = i + 123;
2 d = d * 5;
3 sum = sum / 23.0;
4 flag = flag && (d > 1.0);
```

Kurzform:

```
1 i += 123;
2 d *= 5;
3 sum /= 23.0;
4 flag &&= d > 1.0;
```

Das letzte Beispiel zeigt auch, dass diese Kurzformen nicht nur bei arithmetischen sondern genau so auch bei logischen Operatoren anwendbar sind. Sollte man einmal nicht sicher sein, was eine Kurzform macht, dann macht man sie einfach rückgängig: zuerst die Variable, dann das Zuweisungszeichen, dann die selbe Variable noch einmal, dann den Operator und dann den Rest. Und schon weiss man, was der Ausdruck macht.

Handelt es sich bei dieser Kurzform um einen Seiteneffekt? Ja, denn die Variablen auf der linken Seite werden ja verändert. Handelt es sich um Ausdrücke? Ja, denn es handelt sich immer um Zuweisungen, die allesamt Ausdrücke sind. Also könnte man beispielsweise auch schreiben: `if ( i += 3 ) ... else ...`. Ob allerdings diese Schreibweise immer im Sinne des Erfinders ist, muss jeder für sich selbst entscheiden.

43.4 Pre-/Post- Inkrement/Dekrement

Oftmals ist es so, beispielsweise in `for`-Schleifen, dass man eine Variable gerade mal um eins erhöht oder verringert. In diesen Fällen haben wir bisher immer ausführlich `i = i + 1;` geschrieben. Mit dem Wissen des vorherigen Abschnitts könnten wir auch `i += 1;` schreiben. Aber es geht noch kürzer: `i++;` oder `++i;`. Allgemein kann man sagen: Möchte man den Wert einer einfachen Variablen um eins erhöhen oder verringern, so kann man dies mit zwei vorangestellten oder angehängten Plus- bzw. Minuszeichen machen. Nochmals klar hervorgehoben:

Beispiele: `i++;` `i--;` `++d;` `--c;`

wobei es sich bei `i`, `d` und `c` um Variablen vom Typ `int`, `double` bzw. `char` handelt.

Handelt es sich bei dieser zweiten Kurzform um einen Seiteneffekt? Ja, denn die Variablen auf der linken Seite werden ja verändert. Handelt es sich um Ausdrücke? Ja, denn es

handelt sich immer um Zuweisungen, die allesamt Ausdrücke sind. Also gilt das, was wir im vorherigen Abschnitt über die Verwendung gesagt haben, auch hier.

Jetzt wird's leider etwas kompliziert. Im Falle eines *Pre-* Inkrement/Dekrement wird die verwendete Variable *vor* ihrer Verwendung (als Ausdruck) erhöhnt bzw. verringert. Im Falle des *Post-* Inkrement/Dekrement ist die Sachlage natürlich genau anders herum: die verwendete Variable wird erst *nach* ihrer Verwendung verändert.

„Ist das nicht alles Banane?“ Nicht ganz! Wir erinnern uns: Zuweisungen sind Ausdrücke, die weiter verwendet werden können. Schauen wir auf folgende Beispiele, an deren Ende jeweils die beiden Ausgaben notiert sind.

```
1  i = 3; printf( "%d\n", ++i ); printf( "%d\n", i ); // --> 4 4
2  i = 3; printf( "%d\n", --i ); printf( "%d\n", i ); // --> 2 2
3  i = 3; printf( "%d\n", i++ ); printf( "%d\n", i ); // --> 3 4
4  i = 3; printf( "%d\n", i-- ); printf( "%d\n", i ); // --> 3 2
```

Die Beispiele sollten eigentlich selbsterklärend sein. Sollte es dennoch Unsicherheiten geben, einfach mal die Evaluation so machen, wie wir sie mittels der Zwischenablage oben in Abschnitt 43.1 hatten.

Warnung: Zum Schluss dieses Abschnitts eine kleine, aber dennoch eindringliche Warnung! Dieses Pre-/Post- Inkrement/Dekrement ist eine schöne Tipperleichterung. Aber, es ist ein Seiteneffekt und bei Ausdrücken ist in C explizit festgelegt, dass die Evaluierungsreihenfolge nicht definiert ist (vergleiche auch obigen Abschnitt 43.1). Als Konsequenz sollte man es *tunlichst* (!) vermeiden, den gleichen Seiteneffekt zwei oder mehrmals in einem Ausdruck zu haben! Unbedingt zu vermeiden sind folgende Konstruktionen:

```
1  i = 1; j = (1 + ++i )*(2 + ++i);
2  i = 1; j = (1 + i-- )*(2 + ++i);
```

denn das Ergebnis für `j` ist zwar *deterministisch aber nicht definiert*! Der Wert der Variablen `i` wird zwar jeweils zwei Mal verändert, aber man weiß nicht wann. Definiert ist nur, dass es irgendwann vor der ersten Verwendung bzw. vor dem abschließenden Semikolon passiert. Mehr weiß man nicht. Mögliche Werte für `j` in der ersten Zeile wären: `3*5=15`, `4*4=16` und `4*5=20`. Das liegt daran, wie oben mehrfach erläutert, dass es dem Compiler anheim gestellt ist, wann und in welcher Reihenfolge er die Zwischenablagen (Register) belegt. Hier hilft auch das weitere Verwenden von Klammern nicht weiter. Auch dann kann der Compiler für sich selbst entscheiden, wann er welche Änderung durchführt. Also, in einem Ausdruck *niemals* eine Variable zweimal oder öfters verändern! Auch wenn dies noch so verlockend ist. Das Aufspüren derartiger Fehler ist alles andere als lustig.

43.5 Bedingte Auswertung `?:`

Keine Angst, C ist voller Geheimnisse, es geht weiter mit einer weiteren Kurzform. Des Öfteren ist es so, dass man in Abhängigkeit einer Bedingung ein und derselben Variablen unterschiedliche Werte zuweisen will. Ein sinnfreies Beispiel wäre:

```
1 if ( j == 2 )
2     i = 1 + j*10;
3 else i = 1 - (j - 1)*2;
```

Bei komplexeren Ausdrücken artet dies in sehr viel Schreibarbeit aus, wobei sich beide Varianten oft nur geringfügig voneinander unterscheiden. Genau hierfür gibt es den `?:` Operator. Die Syntax ist: `Ausdruck? True_Ausdruck: False_Ausdruck`. Angwendet auf obiges Beispiel ergibt sich:

```
1 i = (j == 2)? 1 + j*10: 1 - (j - 1)*2;
```

Zur Funktionsweise: Zuerst wird die Bedingung (der Ausdruck vor dem Fragezeichen) ausgewertet. Ergibt dieser einen Wert ungleich null (also logisch wahr), wird er Ausdruck hinter dem Fragezeichen ausgewertet. Im anderen Falle wird der Ausdruck nach dem Doppelpunkt ausgewertet. Der Wert dieses komplexen Ausdrucks ist dann je nach der Bedingung der zweite (`True_Ausdruck`) oder der dritte (`False_Ausdruck`) Ausdruck. In unserem Beispiel wird dieses Ergebnis der Variablen `i` zugewiesen. Häufig ist es sinnvoll bzw. notwendig, den ersten Ausdruck (vor dem Fragezeichen) zu klammern.

Und wieder gilt, dass es sich beim `?:`-Operator um einen Ausdruck handelt, der entsprechend überall stehen darf, wo auch ein anderer Ausdruck stehen darf.

43.6 Logische Ausdrücke

Logische Ausdrücke werden anders behandelt als arithmetische Ausdrücke: C wertet sie von links nach recht aus und bricht das Auswerten ab, sobald das Ergebnis feststeht! *„Was soll das heißen?“* Das ist einfach erklärt: Wenn bei einer und-Verknüpfung der linke Teil bereits `falsch` ergeben hat, wird der rechte Teil einfach nicht mehr ausgewertet, denn `wahr` kann er so oder so nicht mehr werden. Umgekehrt, wird bei einer oder-Verknüpfung der rechte Teil nicht mehr ausgewertet, wenn der linke bereits ein `wahr` ergeben hat, denn der gesamte Ausdruck kann in keinem Fall mehr `falsch` werden.

Auch diese Vorgehensweise macht das Schreiben von Programmen außerordentlich kompakt. Dabei ist wichtig, dass einem klar ist, dass diese Regel immer dann Konsequenzen hat (meist sind diese beabsichtigt), wenn der rechte Teil einen oder mehrere Seiteneffekte hat; diese kommen dann nicht mehr zum Tragen.

43.7 Listen von Ausdrücken

Die meisten von Euch werden sich noch an die `for`-Schleife erinnern, die wir in Kapitel 27 eingeführt haben. Dort haben wir relativ entspannt erwähnt, dass jeder der drei Teile im Schleifenkopf aus mehreren, mit Kommas getrennten Ausdrücken bestehen kann. Beispielsweise können wir schreiben `for( i = 0, summe = 0; ... ; ... )` Dabei bilden die ersten beiden Terme `i = 0` und `summe = 0` eine Liste von Ausdrücken, die nacheinander von links nach rechts abgearbeitet werden; ja, hier ist die Auswertungsreihenfolge definiert.

„Und was soll das jetzt geben?“ Tja, ein tieferes und möglichst vollständiges Verständnis der Programmiersprache C :-) Also, jeder einzelne Term ist eine Zuweisung und damit ein Ausdruck. Was ist dann eine Liste von Ausdrücken? Richtig, wieder ein Ausdruck. Und welchen Wert und Typ hat dieser Ausdruck? Ganz einfach, Wert und Typ sind identisch mit dem letzten Ausdruck. *„Und was soll ich damit anfangen?“* Schauen wir einfach mal auf folgendes Beispiel:

```
1 int i, j, k, l;
2 i = (j = 2, k = 3, l = 4);
3 printf( "i= %d\n", i );
```

Preisfrage: Was wird ausgegeben (sofern diese drei Zeilen innerhalb eines vollständigen Programms stehen würden)? Eines ist klar, die Variablen `j`, `k` und `l` erhalten die Werte `2`, `3` und `4`. So, wie war das mit der Zuweisung nach obiger Erklärung? Der Wert auf der rechten Seite wird in einen Umschlag gepackt und an die Schublade geschickt, die das Ziel der Zuweisung ist. Zusätzlich wird noch ein weiter Umschlag mit dem gleichen Resultat gefüllt und an das Programm zurück geschickt. Ok, bei den ersten beiden Zuweisungen verpufft dieser Umschlag, da ihn niemand verwendet. Aber im dritten Fall geht dieser Umschlag auf die Reise und landet bei der Variablen `i`. Was steckte nochmals in diesem Umschlag? Ach ja, eine 4. Also wird eine 4 ausgegeben. Klar? Einfach mal probieren.

43.8 Diskussion: Seiteneffekte vs. Ausdrücke

Die Wertzuweisung (an eine Variable) ist der klassische Fall eines im Software Engineering erlaubten und erwünschten Seiteneffekts. Neben der Ein- und Ausgabe war's das auch schon. Das bedeutet, dass Seiteneffekte auf der rechten Seite einer Zuweisung *nicht* der Philosophie des Software Engineerings entsprechen. Mit anderen Worten, kurze Anweisungen wie `i = j++ * --k;` lassen zwar das Herz eines C-Programmierers höher schlagen, sind aber aus der Sicht des Software Engineerings keine gute Idee. Der tiefere Grund hierfür liegt wohl darin, dass derartige Konstruktionen zu einem erhöhten Aufwand beim Fehlersuchen führen, zumindest dann, wenn die Programme größer und komplexer werden.

Auf der anderen Seite ist es natürlich so, dass insbesondere diese kleinen Seiteneffekte ein C-Programm wirklich schön kompakt machen können, das dann *unter Umständen* aufgrund

seiner Kürze vielleicht doch wieder einigermaßen übersichtlich ist. Mit anderen Worten: it depends.

Was heißt das nun für Euch? Ja, Ihr sollt auch die besprochenen Kurzformen ausprobieren, einüben und dadurch auch erlernen. Aber tastet Euch ein wenig vorsichtig an dieses Thema heran. Schreibt das Programm ruhig erst einmal ausführlich hin und testet es gründlich. Sollte es fehlerfrei sein, könnt Ihr die einzelnen Anweisungen durch ihre Kurzform ersetzen und schauen, ob es immer noch funktioniert. So wird daraus Spaß und nicht Frust, was schließlich das Wichtigste ist ...

Hinweis: Problematisch werden Seiteneffekte immer im Zusammenhang mit Präprozessor Makros. Nehmen wir an, wir hätten das Makro `#define max(a,b) ((a)>(b))?(a):(b)`. Was passiert bei folgender Anwendung: `m = max( x++, y++ )`? Durch die „stupide“ Textersetzung wird einer der beiden Parameter einmal erhöht, der andere zweimal. Und dies ist in den seltensten Fällen erwünscht. Daher sollte man sowohl bei der Definition als auch Verwendung von „Funktionsmakros“ sehr vorsichtig agieren.

Kapitel 44

Programmierung eigener Funktionen

Funktionen haben wir schon einige kennen gelernt. Neben `main()` und den Ein-/Ausgabefunktionen `scanf()/printf()` gibt es noch zahlreiche mathematische Standardfunktionen wie beispielsweise `sin()`, `cos()`, `sqrt()` und `pow()`. Funktionen kann man aber auch selbst implementieren und sich damit einiges an Arbeit sparen. Darüber hinaus kann dadurch sowohl die Übersichtlichkeit als auch die Änderungsfreundlichkeit deutlich steigen. In Kapitel 38 haben wir gesehen, wie man Makros definieren kann, die wie Funktionen aussehen. Diese sind aber keine richtigen Funktionen, da der Text überall durch anderen Text ersetzt wird. In diesem Kapitel erklären wir, wie man in C Funktionen implementiert, wie sie aufgerufen werden und was auf Hardware-Ebene (CPU/RAM) passiert. Auch hier gilt wieder: nur Mut, einfach mal probieren, der Rechner wird schon nicht kaputt gehen.

44.1 Vorbild: Mathematik

Jeder von Euch kennt das Konzept der Funktion aus der Mathematik. Eine Funktion hat immer einen Namen und ein paar Argumente und liefert einen Wert zurück. Ein schönes Beispiel ist die quadratische Funktion $f(x) = a_2x^2 + a_1x + a_0$. In dieser Definition ist x der freie Parameter, die unabhängige Variable, und a_2, a_1 und a_0 (konstante) Koeffizienten. Nach dieser Definition kann man sie für verschiedene Argumente verwenden. Beispiele sind: $f(0)$, $f(y)$, $f(x+2y)$ und $f(g(x))$. Für jeden dieser Aufrufe liefert die Funktion einen Wert.

Des Weiteren kann man auch mehrdimensionale Funktionen definieren. Beispielsweise definiert $f(x_1, x_2) = \lambda_1 x_1^2 + \lambda_2 x_2^2$ eine quadratische Funktion mit den beiden Variablen x_1 und x_2. Bei Verwendung dieser Funktion müssen wir entsprechend auch zwei konkrete Parameter übergeben.

Soweit sollte alles klar sein. Hinzu kommt, dass die Parameter beim Aufruf einer Funktion ganz anders heißen können als bei der Funktionsdefinition, es können auch konstante Zahlen als Variablenwerte übergeben werden; in beiden Fälle werden die unabhängigen Variablen durch die konkreten Werte ersetzt und alles ist gut.

Im Großen und Ganzen werden Funktionen in C genauso implementiert wie in der Mathematik. Eine Funktion besteht aus einem Namen, einer Zahl von Parametern (den unabhängigen Variablen) und einigen Anweisungen, die den Funktionswert „berechnen“. Und wie in C üblich, muss alles einen Datentyp haben, also auch Funktionen. Anschließend kann man die eine selbst implementierte Funktion aufrufen, „wie man will.“ Natürlich, wer hätte es anders gedacht, müssen wieder die Typen der Parameter und konkreten Werte übereinstimmen, zumindest so halbwegs, denn manchmal kann der Compiler die Typen anpassen, wie wir bereits ausführlich in Kapitel 37.5 besprochen haben.

44.2 Ein Beispiel zum Einstieg

Bevor es formal wird, besprechen wir hier ein kleines, natürlich recht sinnfreies Beispiel, um die wesentlichen Mechanismen auf phänomenologischer Ebene kurz anzusprechen. Schauen wir uns folgendes Programm an:

```
1  #include <stdio.h>
2
3  int sum_mal_2( int a, int b )
4      {
5          int result;
6          result = 2*(a + b);
7          return result;
8      }
9
10 int main( int argc, char **argv )
11     {
12         int r;
13         r = sum_mal_2( 12, -2 );
14         printf( "sum_mal_2( 12, - 2)= %d\n", r );
15     }
```

In den Zeilen 3-8 wird die Funktion `int sum_mal_2( int a, int b )` implementiert. Diese Funktion berechnet $2(a+b)$. Das Ergebnis wird in Zeile 6 in der Variablen `result` abgelegt, die zuvor in Zeile 5 als lokale Variable definiert wurde. In Zeile 7 wird das Ergebnis an die aufrufende Stelle, Zeile 13, zurückgegeben. Wenn wir das Programm ausprobieren, erscheint tatsächlich 20 auf dem Bildschirm.

In Zeile 13 ruft das Hauptprogramm die Funktion `sum_mal_2()` auf. In diesem Moment wird die Abarbeitung des Hauptprogramms an *dieser Stelle* unterbrochen um zunächst die Funktion `sum_mal_2()` abzuarbeiten. Anschließend wird zum Hauptprogramm zurückgekehrt und die Arbeit an der unterbrochenen Stelle wieder aufgenommen. Dieses „Verzweigen“ vom Hauptprogramm in eine Funktion wird am Ende dieses Kapitels nochmals illustriert.

Eine Funktion ist zu Ende, wenn entweder die geschlossene geschweifte Klammer des Funktionsrumpfes erreicht oder die `return`-Anweisung ausgeführt wurde; in erstem Falle ist der Funktionswert undefiniert, was in der Regel unerwünscht ist. An dieser Stelle sei nochmals erwähnt, dass natürlich auch das Hauptprogramm `main()` eine Funktion ist. Desweiteren könnten in der Funktion `sum_mal_2()` weitere Funktionen aufgerufen werden, was nach dem gleichen Schema ablaufen würde.

44.3 Verwendung

Bei Funktionen müssen wir die folgenden drei Dinge auseinander halten:

Funktionsdeklaration:

```
Typ identifier( Parameterliste );
```

Funktionsdefinition:

```
Typ identifier( Parameterliste )
{
   Anweisungen
}
```

Funktionsaufruf:

```
identifier( Parameterliste )
```

Hinweise zur Funktionsdefinition: Bei der Funktions*definition* sollten folgende Punkte beachtet werden:

1. Eine Funktion hat immer einen Typ. Dieser spezifiziert den Typ des Wertes, den die Funktion zurückgibt.
2. Funktionen haben hinter ihrem Namen *immer* ein Paar runder Klammern, auch wenn die Parameterliste leer sein sollte. An den runden Klammern erkennt der Compiler, dass es sich beim Namen nicht um eine Variable sondern um eine Funktion handelt.
3. Die Parameterliste besteht je Parameter immer aus einem Typ und einem Namen. Sollte die Parameterliste aus mehr als einem Parameter bestehen, sind diese mit einem Komma zu trennen.
4. Die Parameter in der Parameterliste bezeichnet man als *formale Parameter* oder auch *Formalparameter*. Erst durch den Funktionsaufruf erhalten sie konkrete Werte.
5. Die eigentlichen Anweisungen, aus denen die Funktion besteht, sind in ein Paar geschweifte Klammern eingeschlossen. Diese müssen im Gegensatz zu den Kontrollstrukturen immer vorhanden sein, auch wenn die Funktion aus nur einer einzigen oder gar keiner Anweisung besteht.
6. Hinter der schließenden runden Klammer im Funktionskopf (der aus Typ, Klammern und Parameterliste besteht) darf *kein* Semikolon stehen.

7. Im Funktionsrumpf dürfen beliebig viele Variablen deklariert werden, die auch *lokale* Variablen genannt werden. Diese lokalen Variablen werden bei jedem Funktionsaufruf erneut angelegt, können nur innerhalb dieser Funktion verwendet werden und verschwinden nach Beendigung der Funktion wieder aus dem Arbeitsspeicher.
8. Durch die Anweisung `return Wert` wird die Funktion verlassen und ein Wert zurückgegeben. Ohne diese Anweisung ist der Rückgabewert der Funktion *undefiniert*.
9. Man sagt auch, dass durch die `return`-Anweisung ein Wert an die aufrufende Stelle zurückgegeben wird.

Hinweise zur Funktionsdeklaration: Bei der Funktions*deklaration* sollten folgende Punkte beachtet werden:

1. Die Funktionsdeklaration besteht nur aus dem Funktionskopf (Typ, Name, Parameterliste) und einem abschließenden Semikolon. Das Semikolon muss dort unbedingt stehen.
2. Die Funktionsdeklaration hat *keinen* Funktionsrumpf und somit weder geschweifte Klammern noch Anweisungen.

Hinweise zum Funktionsaufruf: Beim Funktions*aufruf* sollten folgende Punkte beachtet werden:

1. Der Funktionsaufruf besteht aus dem Namen der Funktion, einer runden, öffnenden Klammer, den Parametern und einer runden, schließenden Klammer.
2. Die Parameter beim Funktionsaufruf nennt man auch *aktuelle Parameter*.
3. Die runden Klammern müssen vorhanden sein, auch wenn die Funktion keinen Parameter besitzt.

Funktionsdeklaration vs. Funktionsdefinition: Bisher haben wir immer gesagt, dass eine Variable erst benutzt werden darf, wenn sie auch definiert ist. In der Programmiersprache C ist dies bei Funktionen nicht so: eine Funktion darf auch ohne vorherige Definition einfach verwendet werden. Der Compiler geht dann davon aus, dass die Funktionsdefinition irgendwann später kommen wird, oder sich gar in einer anderen Datei befindet. Ferner nimmt der Compiler in diesem Falle an, dass sowohl die Funktion als auch alle ihre Parameter vom Typ `int` sind. Sollte dies nicht der Fall sein, gibt's früher oder später ein Problem. Aus diesem Grund gibt es die Funktions*deklaration*. Durch sie macht man dem Compiler klar, um was für eine Funktion es sich handelt, von welchem Typ sie ist und welchen Typ ihre Parameter haben. Für den Anfang empfehlen wir folgende Vorgehensweise: Erstens, eine Funktion erst zu definieren und sie dann zu verwenden. Sollte dies aus irgendeinem Grund nicht möglich sein, schreibe man den Funktionskopf gefolgt von einem Semikolon möglichst weit oben in den Quelltext. Dann weiß der Compiler Bescheid.

44.4 Aufruf und Abarbeitung von Funktionen

Für viele Programmieranfänger ist die Abarbeitung von Funktionen ein Rätsel erster Ordnung. Dabei ist dies sehr einfach, wie wir in diesem Abschnitt sehen werden. In Kapitel 36 sind wir im Detail darauf eingegangen, wie die CPU ein Programm abarbeitet. Insbesondere haben wir besprochen, dass es dafür den Program Counter (PC) gibt, der jeweils die Adresse der nächsten Maschineninstruktion beinhaltet. Dies ist auch nach wie vor der Fall.

Was passiert nun bei einem Funktionsaufruf? In diesem Fall wird die Abarbeitung des Programms unterbrochen. Die CPU „merkt“ sich diese Stelle und beginnt die angegebene Funktion abzuarbeiten. Wenn sie damit fertig ist, kehrt sie an die Stelle zurück, die sie sich vorher gemerkt hat.

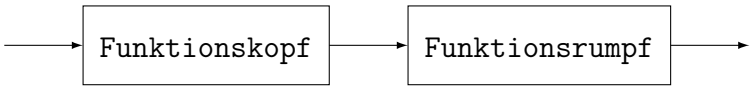
Sollte es in der aufgerufenen Funktion einen weiteren Funktionsaufruf geben, kommt dieses Prinzip auch dort zur Anwendung: Merken der aktuellen Stelle im Programm, Sprung zur angegebenen Funktion und nach Beendigung der Funktion Rückkehr zur alten Stelle. Dieses verschachtelte Aufrufen weiterer Funktionen kann quasi beliebig oft wiederholt werden.

Dieses Prinzip haben wir völlig unbemerkt bei jeder einzelnen Ein- bzw. Ausgabeanweisung erlebt: Jeder Aufruf von `scanf()` und `printf()` ist ein Funktionsaufruf. Das Hauptprogramm (`main()`) wird an dieser Stelle unterbrochen, die jeweilige Funktion wird vollständig abgearbeitet und anschließend wird im Hauptprogramm an der alten Stelle weiter gemacht.

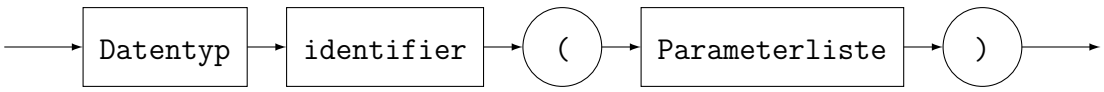
Dieser Vorgang ist am Ende dieses Kapitels noch einmal detailliert dargestellt.

44.5 Syntaxdiagramme

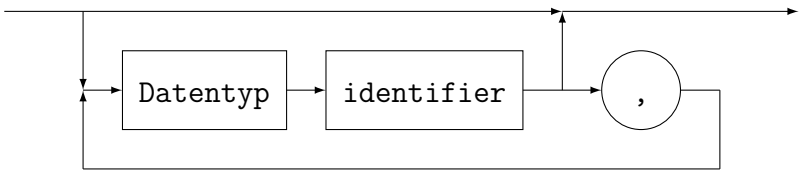
Funktionsdefinition



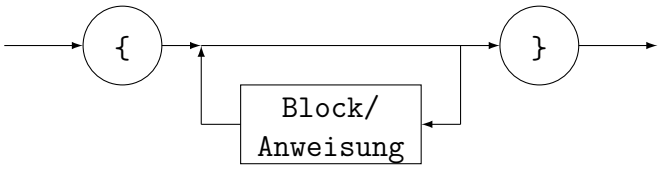
Funktionskopf



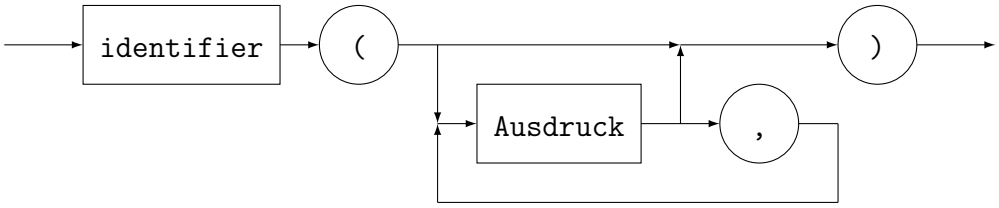
Parameterliste



Funktionsrumpf

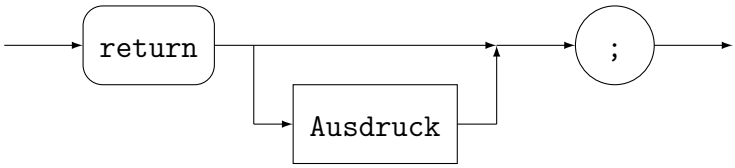


Funktionsaufruf



Die `return`-Anweisung hatten wir in Kapitel 23 zwar schon erwähnt aber noch nicht weiter spezifiziert. Sie kann überall dort stehen, wo auch eine der anderen Anweisungen stehen darf. Hier nun das einfache Syntaxdiagramm:

return-Anweisung



44.6 Anforderungen an den Speicherbedarf

Zunächst greifen wir nochmals ein paar Hinweise aus Abschnitt 44.3 auf:

1. Sowohl die formalen Parameter als auch die lokalen Variablen werden bei jedem Funktionsaufruf erneut angelegt und verschwinden mit dem Ende der Funktion wieder aus dem Arbeitsspeicher.
2. Beim Aufruf einer Funktion wird der „normale“ Verarbeitungsfluss unterbrochen, die aktuelle Position an geeigneter Stelle vermerkt, in die Funktion verzweigt und nach Beendigung der Funktion an die gemerkte Position zurückgekehrt.
3. Funktionen können beliebig ineinander verschachtelt aufgerufen werden.

Aus diesen drei Punkten resultiert, dass der gesamte notwendige Speicherbedarf nicht im voraus ermittelt werden kann, denn der Compiler kann nicht immer ermitteln, wie oft welche Funktionen wie verschachtelt aufgerufen werden. Mit anderen Worten, die Speicherplatzreservierung muss dynamisch zur Laufzeit passieren: Immer wenn eine Funktion aufgerufen wird, wird für eben diese der notwendige Speicherplatz organisiert.

Glücklicherweise können wir uns noch folgenden Umstand zu Nutze machen: Wenn eine Funktion `f1()` eine Funktion `f2()` aufruft, wird definitiv erst `f2()` beendet, bevor `f1()` beendet wird. Mit anderen Worten: Funktionen werden immer in umgekehrter Reihenfolge beendet wie sie gestartet werden. Das ist übrigens in der Mathematik auch so ;-) Da die zuletzt erzeugten Speicherbereiche wieder zuerst aus dem Arbeitsspeicher entfernt werden müssen, benötigen wir für die Abarbeitung von Funktionen einen Stack (engl. für Kellerspeicher), wie wir im nächsten Abschnitt erklären.

44.7 Abarbeitung mittels Stack Frame

Aufgrund der im vorherigen Abschnitt erläuterten dynamischen Speicherplatzanforderungen werden alle notwendigen Variablen einer Funktion, formale Parameter und lokale Variablen, grundsätzlich auf dem Stack[1] angelegt. Hierzu besitzt die CPU ein spezielles Register, den *Stack Pointer* (SP), den wir bereits in Kapitel 36 erwähnt haben. Bei jedem Funktionsaufruf wird mittels des Stack Pointers ein entsprechendes Segment auf dem Stack erzeugt, das nach dem Funktionsende wieder entfernt wird. Die hierfür notwendigen Maschineninstruktionen werden vom Compiler eingebaut. Und wie schon in Kapitel 40 erwähnt, wächst der Stack von hohen Adressen, beispielsweise `0xFFFFFFFF`, zu kleineren nach unten [2].

Die folgenden Erläuterungen beziehen sich auf das Beispiel, das wir in Abschnitt 44.2 besprochen haben; die Konzepte sind aber natürlich allgemeingültig für C. Für jede Funktion verwaltet der Compiler einen sogenannten *Stack Frame*, der der Reihe nach aus den formalen Parametern, dem Rückgabewert (der Funktion), dem aktuellen Program Counter und den lokalen Variablen besteht[3]. In unserem Beispiel sieht dieser Stack Frame wie folgt aus:

Stack Frame: Funktion: sum_mal_2(int a, int b)

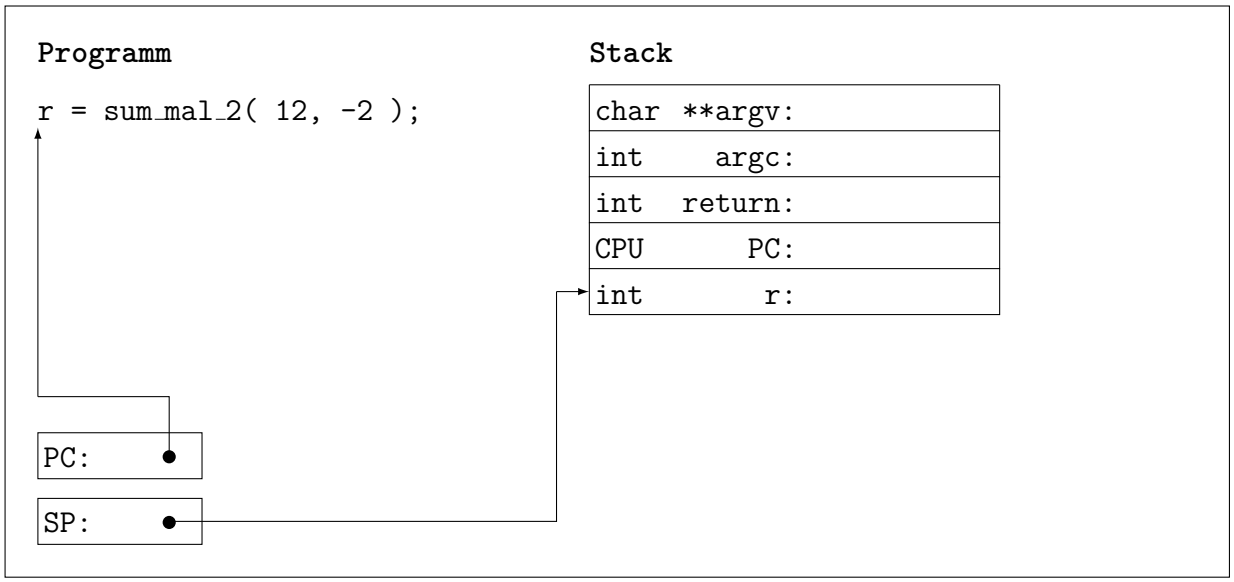
`int b:`
`int a:`
`int return:`
`CPU PC:`
`int result:`

Die Adressen innerhalb des Stack Frames sind keine richtigen Speicheradressen sondern nur kleine relative Abweichungen (Offsets) vom Stack Pointer. Die CPU berechnet jedesmal die richtige Speicheradresse aus Offset und Stack Pointer. Da sie dafür ein eigenes Rechenwerk hat, geht dies in einem Taktzyklus; bei einer Taktfrequenz von 1 GHz ist dies 1 ns.

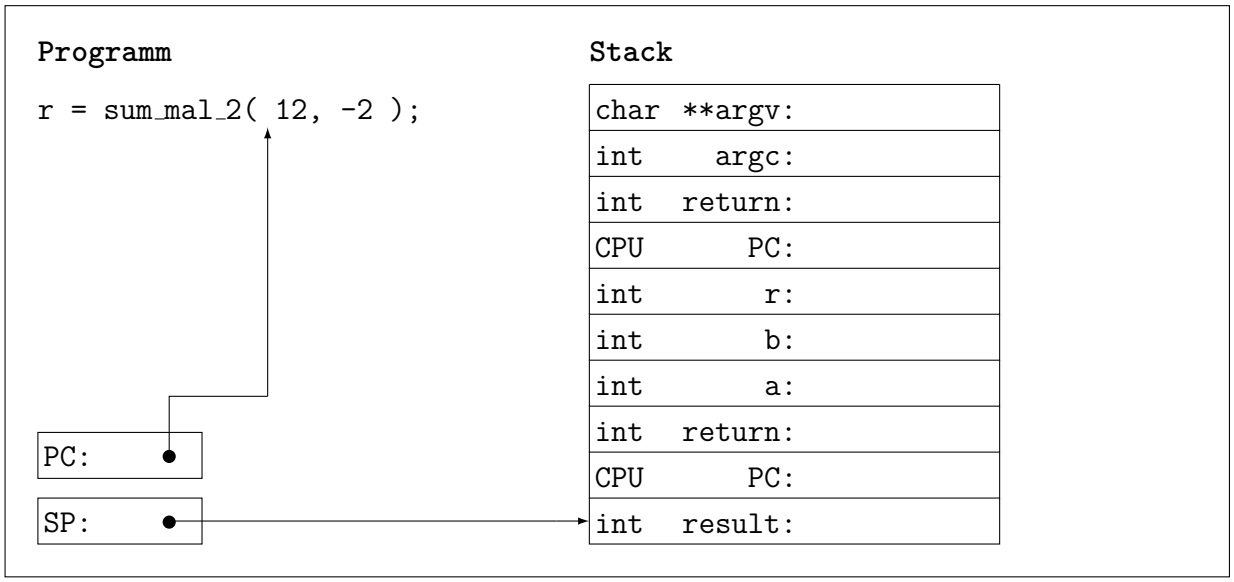
Wie gesagt, ein derartiger Stack Frame wird bei *jedem* Funktionsaufruf generiert. Im Folgenden nehmen wir die Anweisung `r = sum_mal_2( 12, -2 )` im Einzelnen auseinander

[1] Von dieser Regel gibt es eine Ausnahme, die wir in Kapitel 56 besprechnen.
[2] Je nach Compiler und Hardware kann die höchste verfügbare Adresse von `0xFFFFFFFF` abweichen.
[3] Je nach Compiler und Hardware werden noch weitere Daten im Stack Frame abgelegt.

und besprechen, was im Arbeitsspeicher passiert. Aus Gründen der Übersichtlichkeit verzichten wir auf konkrete Adressen. Als „alten“ Wert für den Program Counter nehmen wir die nächste Programmzeile zu der *nach* dem Ende des Funktionsaufrufes zurückgekehrt wird. Vor dem Funktionsaufruf sieht der Stack wie folgt aus:

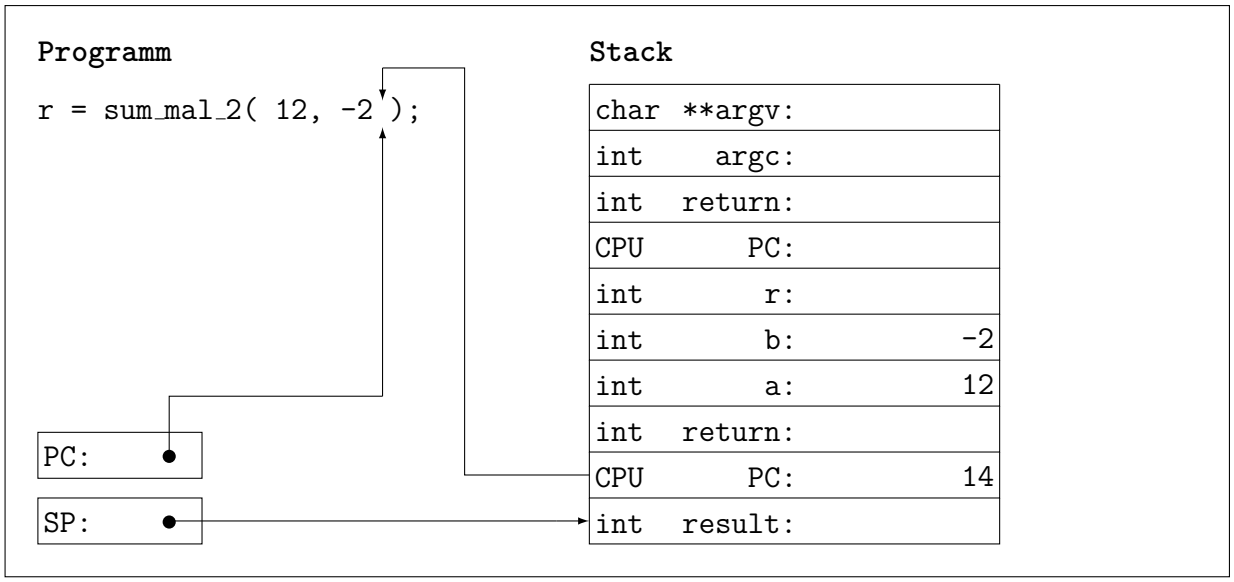


Wir sehen, wie sich die Variablen des Hauptprogramms `main()` bereits auf dem Stack befinden. Als erste Aktion wird zunächst der Stack Frame der Funktion `sum_mal_2()` angelegt. Diese Kopie sieht genau so aus, wie wir bereits weiter oben illustriert haben. Ferner wird durch das Anlegen dieses Stack Frames der Stack Pointer entsprechend nach unten verschoben, sodass er anschließend an das untere Ende des soeben erzeugten Frames zeigt. Dies haben wir im Bild oben auf der nächsten Seite veranschaulicht.

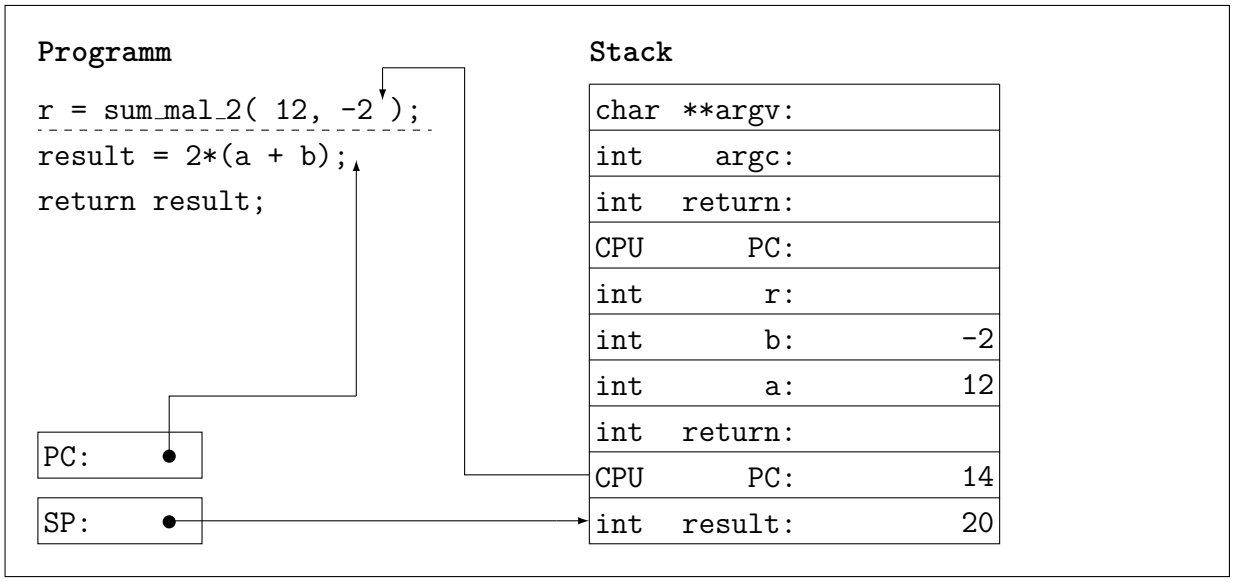


Als nächstes werden die aktuellen Parameter der Reihe nach ausgewertet und den formalen Parametern des Stack Frames zugewiesen. In unserem Fall wird die Funktion `sum_mal_2()` nur mit Konstanten aufgerufen. Entsprechend bekommen die beiden formalen Parameter die Werte `a = 12` und `b = -2`.

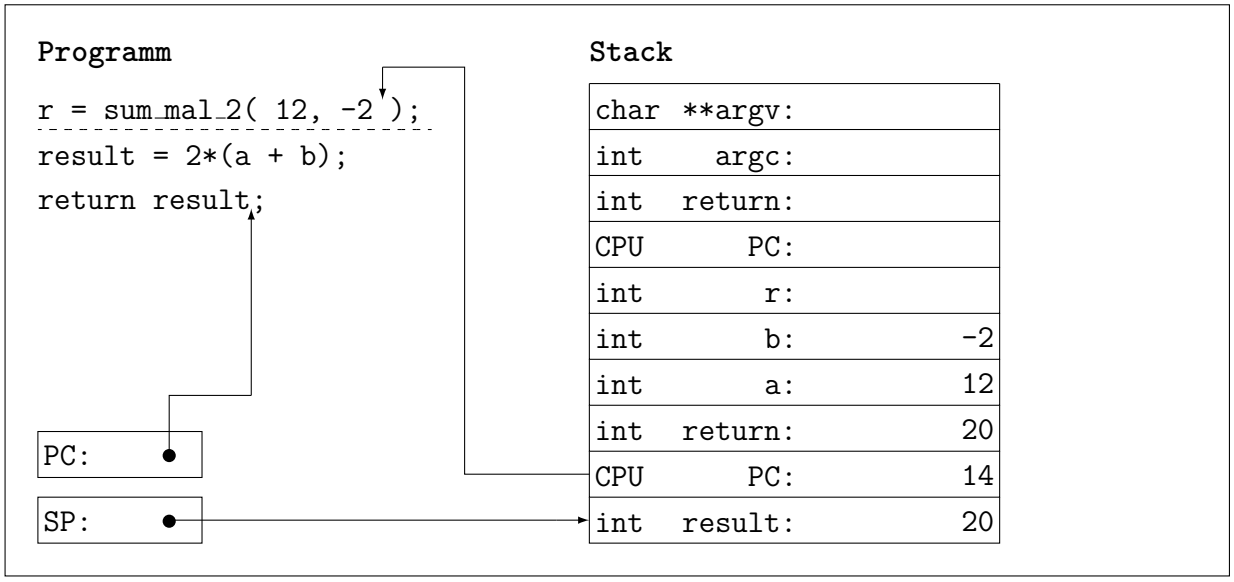
Da das Programm nach dem Funktionsaufruf mit der Zeile `14` fortgesetzt werden muss, wird genau diese „Rücksprungadresse“ auf dem Stack gerettet. Folgerichtig wird der entsprechende Eintrag im Stack Frame auf genau diesen Wert gesetzt. Hierhin wird die CPU zurückkehren, wenn sie mit der Abarbeitung des Funktion `sum_mal_2()`fertig ist und das Ergebnis des Funktionsaufrufs der Variablen `r` zugewiesen hat. Es sei nochmals daran erinnert, dass die Programmzeile als Wert des Program Counters nur eine starke Vereinfachung ist, um den schwierigen Sachverhalt nicht noch unnötig zu verkomplizieren. Der resultierende Stack Frame sieht wie folgt aus:



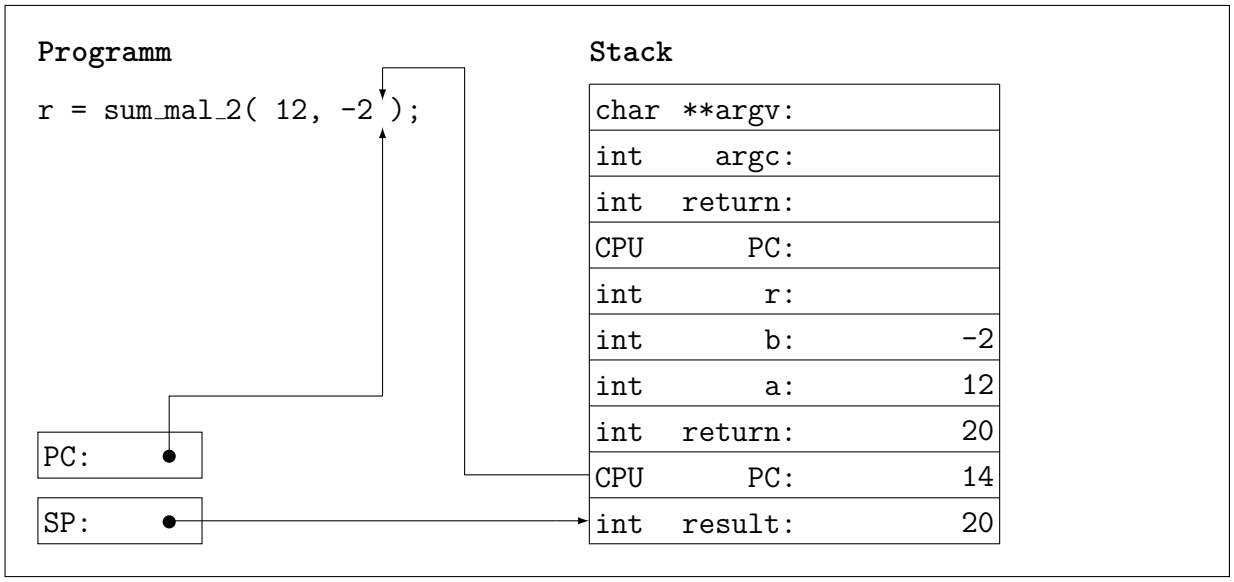
Nun beginnt die CPU mit der Abarbeitung der Funktion `sum_mal_2()`. Der erste Schritt beim Abarbeiten der Funktion `sum_mal_2()` besteht darin, das Zwischenergebnis `2*(a+b)` zu berechnen. Dieses Ergebnis wird anschließend in der Variablen `result` abgelegt, die sich ebenfalls im Stack Frame befindet, da sie eine lokale Variable dieser Funktion ist. Der durch diese Aktionen veränderte Stack sieht wie folgt aus:



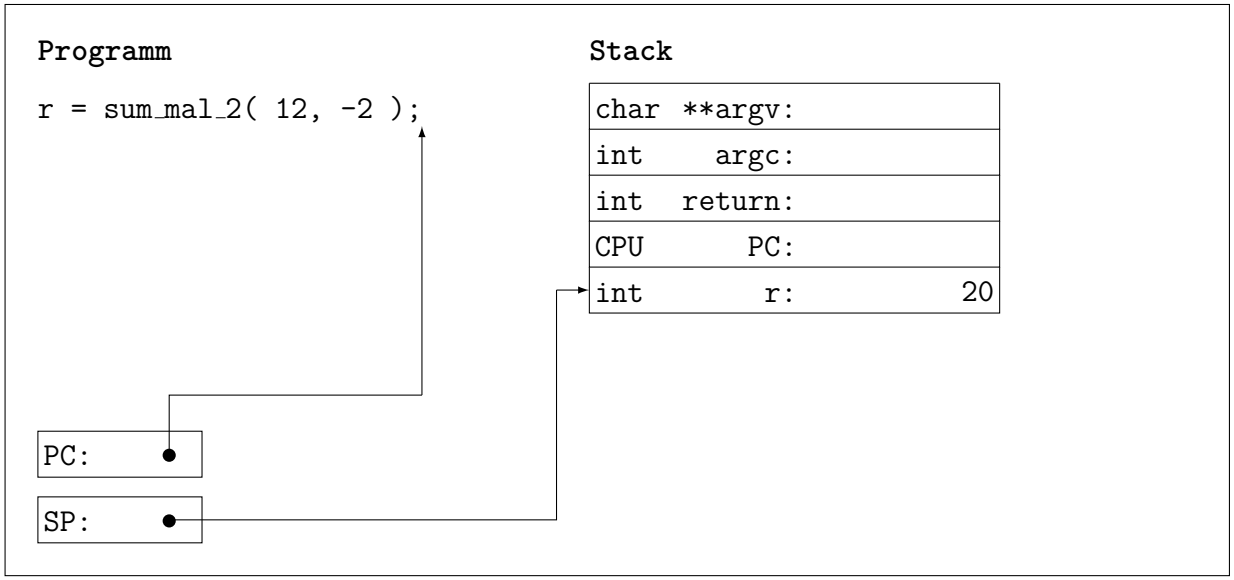
Anschließend wird die `return result`-Anweisung ausgefürt. Das Ausführen dieser Anweisung veranlasst die CPU dazu, den Wert des Rückgabewertes zu nehmen und an die reservierte Speicherstelle des Stack Frames zu kopieren. In unserem Beispiel besteht der Rückgabewert lediglich aus dem Ausdruck `result`. Entsprechend wird der Inhalt dieser Variablen genommen und an die entsprechende Stelle kopiert. Diese Stelle ist im Bild mit „`int return`“ angeschrieben. Nun ist der Rückgabewert an seiner richtigen Stelle:



Mit Beendigung der `return`-Anweisung wird die Funktion `sum_mal_2()` verlassen und der Program Counter wird auf die alte Stelle „Programmzeile“ `14` zurückgesetzt. Nun „weiß“ die CPU, an welcher Stelle sie das vorher unterbrochene (Haupt-) Programm fortsetzen soll:



Im Hauptprogramm wird jetzt das Ergebnis der `return`-Anweisung verarbeitet, d.h. der Variablen `r` zugewiesen, sowie der Stack Frame wieder abgebaut.



Und wir sind fertig. Nun sieht auch der Stack wieder aus wie am Anfang.

44.8 Zusammenfassung

In diesem Kapitel haben wir gelernt, dass Funktionen eigenständige Unterprogramme sind. Sollte eine Funktion aufgerufen werden, wird der normale Programmablauf unterbrochen. Nach Beendigung der Funktion wird an die ursprüngliche Stelle zurückgekehrt und dort mit der Abarbeitung weiter gemacht (als wenn nichts gewesen wäre).

Bei jedem Funktionsaufruf wird ein Stack Frame angelegt, der die formalen Parameter

sowie alle lokalen Variablen enthält. Zu Beginn des Funktionsaufrufs erhalten die formalen Parameter die Werte der aktuellen Parameter zugewiesen. Da jede Funktion ihren eigenen Stack Frame besitzt, arbeitet sie auch immer auf ihren eigenen lokalen Variablen. Da diese am Ende des Funktionsaufrufs *immer verschwinden,* haben Änderungen dieser Variablen *keinen* Einfluss auf die Werte von Variablen, die außerhalb der Funktion definiert wurden. Nur über den Rückgabewert einer Funktion kann man etwas an die aufrufende Stelle im Programm zurückgeben. Diesen Übergabemechanismus, bei dem eine Funktion immer nur auf lokalen Kopien der aktuellen Parameter arbeitet, nennt man *Call-by-Value*. Diese Sichtweise auf Funktionen entspricht auch der Philosophie des Software Engineerings, nach der sich eine Funktion bei gleichen Parametern auch immer gleich verhalten sollte.

44.9 Ausblick: Funktionen und Arrays

In diesem Kapitel haben wir sehr bewusst nur einfache Variablen an eine Funktion übergeben. Man kann aber auch ganze Arrays übergeben. Doch muss man dann *unbedingt* auch die Größe dieses Arrays als Parameter übergeben, da die Funktion keine Chance hat, die Größe eines Arrays zu ermitteln. Im folgenden Programmbeispiel bekommt die Funktion `set_to_1()` ein Array übergeben, dessen Elemente sie alle auf den Wert `1` setzt:

```
1  #define SIZE    5
2
3  int set_to_1( int a[], int size )
4      {
5          int i;
6          for( i = 0; i < size; i++ )
7              a[ i ] = 1;
8          return 1;
9      }
10
11 int main( int argc, char **argv )
12     {
13         int arr[ SIZE ];
14         set_to_1( arr, SIZE );
15     }
```

Wichtiger Hinweis: Bei Verwendung von Arrays operiert die Funktion nicht auf einer lokalen Kopie dieses Arrays sondern auf dem Original. Dieser Seiteneffekt widerspricht den gerade eben ausgeführten Erläuterungen zum Thema Software Engineering. Die genaue Funktionsweise und die verwendeten Mechanismen erläutern wir in Kapitel 47. Auch wenn jetzt einige sagen: *„Das ist doch klar, Call-by-Reference!“* dann ist dies *grundsätzlich falsch,* auch wenn es immer wieder wiederholt wird. Die Programmiersprache C kennt nur einen Parameterübergabemechanismus und der heißt Call-by-Value! Ausrufezeichen!

Kapitel 45

Zeiger und Adressen

In diesem Kapitel gehen wir darauf ein, wie Zeiger und Adressen innerhalb eines C-Programms verwendet werden. Zeiger und Adressen sind insbesondere im Zusammenhang mit Arrays und Funktionen von elementarer Bedeutung. Entsprechend gibt es in C auch einen Datentyp *Zeiger* (engl. *Pointer to*), von dem wir auch Variablen deklarieren und definieren können, wie wir es von jedem anderen Datentyp her gewohnt sind. Ferner kann man mit Adressen auch rechnen, was allerdings eine gewisse Übung voraussetzt. Die von den Sprachentwicklern festgelegte Notation stellt eine kleine Barriere dar, da ein und das selbe Zeichen, nämlich der Stern *, sowohl in Definition als auch Applikation (Anwendung) verwendet wird, er aber in beiden Fällen eine *komplett unterschiedliche* Bedeutung hat.

Dieses Kapitel ist nun so aufgebaut, dass wir am Anfang einen kleinen historischen Rückblick geben, um die C-Design-Entscheidungen besser einordnen zu können. Dann wiederholen wir nochmals einige Kernkonzepte, die beim Zugriff auf Variablen wichtig waren, und erklären dann, wie man Zeiger in C verwendet. Vor dem Weiterlesen sollte sich jeder Leser nochmals vergewissern, dass er die Inhalte der beiden Kapitel 36 und 37 noch kennt und diese auch anwenden kann. Falls nicht, einfach nochmals durcharbeiten, denn der dortige Stoff ist einfach Voraussetzung für dieses Kapitel!

45.1 Historischer Rückblick

Die Behandlung von Zeigern und Adressen in C ist für den Programmieranfänger sowie den leicht Fortgeschrittenen eher etwas mühsam. Um die damals getätigten Design-Entscheidungen besser nachvollziehen zu können, gibt dieser Abschnitt einen kurzen historischen Rückblick.

C entstand in einer Zeit, in der man aufgrund angenommener Performanzprobleme Betriebssysteme ausschließlich in Assembler implementierte. Im Rahmen des damaligen Unix-Projektes entstand die Idee, eine Programmiersprache zu entwickeln, die man auf der einen

Seite als Hochsprache ansehen kann, einem auf der anderen Seite aber sehr Hardware-nahe Konzepte zur Verfügung stellt, die die Entwicklung sehr effizienter Programme erlauben.

Sowohl die Sprachentwickler als auch die damaligen C-Programmierer waren Oberexperten auf ihrem Gebiet. Sie wussten mit Sicherheit bei jeder einzelnen Anweisung, was der Compiler daraus macht und wie sie von der Hardware abgearbeitet wird. Für diese Experten war möglichst kurzer, kompakter Code gefragt, der alles andere als anfängerfreundlich ist.

Eines dieser Konzepte ist das Bestimmen, Verwenden und Berechnen von Adressen. Dieser Ansatz kann, wie oben schon gesagt, zur Entwicklung sehr effizienter Programme verwendet werden. Für Experten sind diese Adressberechnungen ein Klacks und Alltag, für den Anfänger mühsam und fehleranfällig.

Im Zusammenhang mit dem Unix-Kernel war die Programmiersprache C so erfolgreich, dass sie ihr eigenes Opfer wurde: Mittlerweile ist C überall anzutreffen, insbesondere wenn es um die Hardware-nahe Programmierung geht, die für Elektrotechniker eine gewisse Bedeutung hat. Insofern ist es so, dass auch Ihr hier durch müsst, auch wenn C zum Erlernen des Programmierens eher ungünstig ist. Aber die gute Nachricht ist: bisher ist keiner ertrunken, alle haben es geschafft. Und ein Trost bleibt, in vielen anderen Programmiersprachen kann man gar nicht die Adressen von Variablen und Funktionen bestimmen ...

45.2 Verwendung von Variablen: Ein Rückblick

In den beiden Kapiteln 36 und 37 haben wir bezüglich der Zugriffe auf Variablen unter anderem folgendes besprochen:

1. Die CPU muss die Speicheradresse einer Variablen kennen, damit sie entweder deren Wert verwenden (lesen) kann oder ihr einen neuen Wert zuweisen (schreiben) kann.
2. Nur durch die Typ-Information kann der Compiler *genau* wissen, wie die Bitmuster zu interpretieren sind, welche Anweisungen er für die CPU auswählen und wie viele Bytes er zwischen Arbeitsspeicher und CPU hin und her transferieren muss. Daher: der Compiler muss immer *genau* wissen, von welchem Typ die Operanden sind.
3. Aufgrund der Typ-Problematik macht der Compiler aus so einer einfachen Anweisung wie `int i = 5` folgendes: $\texttt{i}_{\texttt{int}} =_{\texttt{int}} \texttt{5}_{\texttt{int}}$.
4. Compiler und CPU kennen die Adresse jeder Variablen oder können diese jederzeit sehr leicht ausrechnen.
5. Die Adresse einer Variablen bzw. Funktion kann man sehr leicht mittels des Adressoperators `&` ermitteln. Beispiel: `& i` bestimmt die Speicheradresse der Variablen `i`.

Eine grundlegende Idee des Designs der Programmiersprache C ist nun wie folgt: Wenn Compiler und CPU sowieso die Adressen kennen (müssen), kann man sie doch auch einfach direkt innerhalb des C-Programms verwenden.

45.3 Definition von Zeigern: ein erstes Beispiel

Dieser Abschnitt präsentiert ein kleines Beispiel vorweg, damit Ihr wisst, wie diese Zeiger aussehen und wie man eine derartige Definition am besten liest.

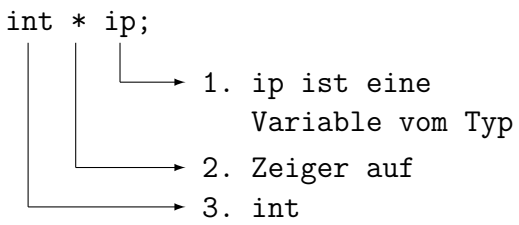
Zeigervariablen (Pointer) definiert man einfach, in dem man *vor* den Variablennamen ein Sternchen schreibt. Beispielsweise definiert

```
int * ip;
```

eine Variable namens `ip`, die jetzt keine „normalen“ Zahlen wie `-1` und 4711 mehr aufnehmen kann, sondern Speicheradressen von Variablen, die vom Typ `int` (und keinem anderen (!)) sind. Mit anderen Worten: An dieser Stelle darf sich keine `char`, `double` oder anders als `int` geartete Variable befinden. Dieser Sachverhalt ist essentiell!

Lesen einer Zeigerdefinition:

Eine derartige Definition ließt man am besten von innen (Variablenname) nach außen (ganz links). Und sollte rechts nichts stehen (außer einem Komma oder Semikolon), schaut man links und findet das Sternchen ...



45.4 Beispielhafte Verwendung

Um sich an Zeiger zu gewöhnen, vergegenwärtigt man sich am besten immer wieder die Notation, wie sie in der abstrakten Programmierung und anderen Hochsprachen wie beispielsweise Pascal und Modula-2 verwendet wird. Es hilft! Also bei der Definition immer: *Pointer to* oder *Zeiger auf*.

C-Syntax	**Abstrakte Programmierung**
`int i;`	`Variable: Integer: i`
`int * ip;`	`Variable: Typ Pointer to Integer: ip`
`ip = & i;`	`setze ip = Adresse von i`
`*ip = 2;`	`setze ip@ = 2 // ip dereferenziert`

Eigentlich nicht so schwer, oder? Im Folgenden besprechen wir jede einzelne Zeile:

`int i:`
: Inzwischen sollte jedem klar sein, dass hier ein Variable definiert wird, und zwar die Variable `i` vom Typ `int`. Falls nicht, gehe zurück auf Los, ziehe keine 4000, gehe nicht ins Hotel, ...

`int *ip:`
: Hier wird eine Zeiger-Variable definiert, wie wir es gerade eben im vorherigen Abschnitt besprochen haben. Zur Wiederholung: `ip` ist eine Variable vom Typ Zeiger

auf `int`. Entsprechend kann diese Variable Werte annehmen, die Adressen im Arbeitsspeicher sind, an deren Stelle sich ein `int` befindet. Dabei ist beides wichtig: es muss eine Adresse sein *und* dort muss sich ein `int` befinden.

`ip = & i;`
Den rechten Teil hatten wir schon seit unserem ersten Programm (Kapitel 7). Durch die Konstruktion mit dem `&`-Zeichen bestimmen wir die Adresse der Variablen `i` im Arbeitsspeicher. Diesen Wert, also die Speicheradresse einer `int`-Variablen, weisen wir wie gewöhnlich der Zeigervariablen `ip` zu. Letztlich steht dort nichts anderes als eine gewöhnliche Zuweisung wie `j = 2`; nur haben sich jetzt die beteiligten Werte und Typen verändert.

`*ip = 2;`
Diese Zeile ist das wirklich neue in diesem Abschnitt. Rechts steht eine `2`. Diese wird wieder in den Briefumschlag gepackt. Links steht jetzt aber nicht mehr `ip` sondern `*ip`. Entsprechend wird die `2` nicht an die Stelle geschickt, an der die Variable `ip` zu finden ist, sondern an die Stelle, auf die durch `ip` verwiesen wird.

Innerhalb von Ausdrücken (also *nicht* von Definitionen) nennt man dieses Sternchen auch *dereferenziert*. Man bezeichnet diese Konstruktion auch als *einfach indirekten* Speicherzugriff.

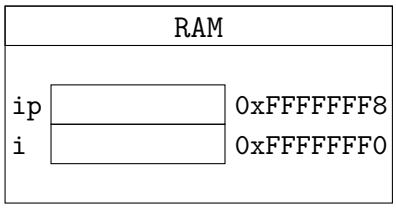
45.5 Funktionsweise des Beispiels

Da wir davon ausgehen, dass nicht jeder die Erklärungen des vorherigen Abschnittes vollständig verstanden hat, schauen wir uns die Funktionsweise nochmals auf drei unterschiedliche Arten an. Eine dieser Erklärungen müsst Ihr einfach verstehen, hilft alles nichts. Das Lehrpersonal steht euch für weitere Fragen gerne zur Verfügung und geht mit euch auch gerne nochmals die Erklärungen gemeinsam durch.

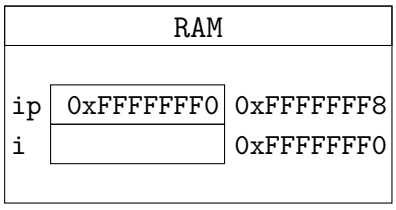
Erklärungsversuch Speicherbild:

Versuchen wir zunächst, uns nochmals klar zu machen, was im Arbeitsspeicher vor sich geht. Die folgenden drei Bilder zeigen die jeweiligen Inhalte des Arbeitsspeichers nach Ausführung der einzelnen Anweisungen:

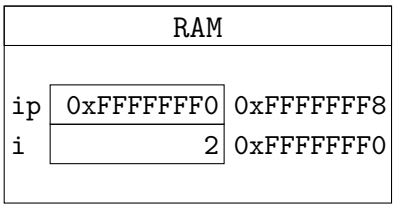
`Variablendefinition`



`Nach ip = & i;`



`Nach *ip = 2;`



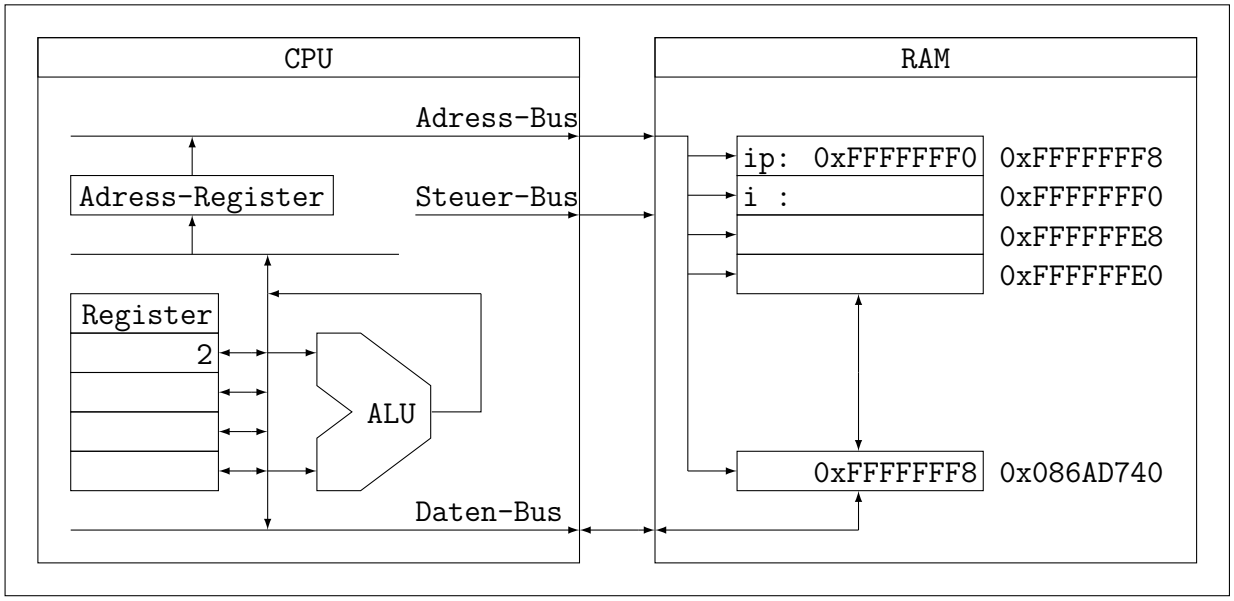
Im ersten Bild sehen wir, dass nach der Definition die beiden Variablen `i` und `ip` (rein zufällig) die Adressen `0xFFFFFFF0` und `0xFFFFFFF8` haben. Durch die anschließende Zuweisung `ip = & i` erhält die Variable `ip` den Wert `0xFFFFFFF0`, also diejenige Adresse,

unter der wir die Variable `i` im Arbeitsspeicher finden. Das dritte Bild zeigt, dass durch das einfache Dereferenzieren `* ip` der Wert `2` nicht der Variablen `ip` zugewiesen wird, sondern an die Speicherstelle gelangt, die in der Variablen `ip` angegeben ist. Würden wir jetzt die Variable `i` ausgeben, käme tatsächlich `2` heraus.

Erklärungsversuch CPU:

Auf dieser Ebene schauen wir uns genau an, wie die CPU Zeigerzugriffe abarbeitet.

`Reduzierter Aufbau von CPU und RAM nach ip = & i`



Nun gehen wir das ganze nochmals durch und schauen dabei, was die CPU machen muss, um die Anweisung `* ip = 2` auszuführen. Diese Betrachtungen auf der Ebene der CPU und Register sind naturbedingt sehr anstrengend und voller Details. Daher sollte jeder Leser die folgenden Erläuterungen selbstständig auf einem Blatt Papier nachvollziehen.

Zur Vorbereitung sollte sich auch jeder Leser nochmals die folgenden Dinge klar machen: Wo (in welchem Adressbereich) befindet sich der Maschinencode? Wo (in welchem Adressbereich) befindet sich die Daten (Variablen)? Wo befinden sich Konstanten (Werte und Adresse), die die CPU für die Ausführung ihrer Instruktionen benötigt?

Wie wir gleich sehen werden, muss die CPU durch den *indirekten* Speicherzugriff, das Adress-Register einmal mehr laden als dies bei Variablenwerten der Fall ist. Wie in Kapitel 36 beschrieben muss die CPU erst einmal die Adresse unseres Zeigers in das Adress-Register laden. Zu diesem Zeitpunkt zeigt der Program Counter (PC) auf die entsprechende Adresse `PC == 0x086AD740`. Es ergeben sich die folgenden Schritte:

	Adress-Bus	Daten-Bus	Adress-Register	Erläuterung
1	→ `0x086AD740`	.....	.....	RAM-Adresse anwählen
2	`0x086AD740`	→ `0xFFFFFFF8`	.....	RAM sendet Zeigeradresse
3	`0x086AD740`	`0xFFFFFFF8`	→ `0xFFFFFFF8`	Zeigeradresse laden
4	→ `0xFFFFFFF8`	`0xFFFFFFF8`	`0xFFFFFFF8`	*Adresse* von `ip` an RAM
5	`0xFFFFFFF8`	→ `0xFFFFFFF0`	`0xFFFFFFF8`	*Inhalt* von Zeiger `ip`
6	`0xFFFFFFF8`	`0xFFFFFFF0`	→ `0xFFFFFFF0`	Inhalt (Adresse von `i`) laden
7	→ `0xFFFFFFF0`	`0xFFFFFFF0`	`0xFFFFFFF0`	Adresse von `i` an RAM
8	`0xFFFFFFF0`	→ `2`	→ `0xFFFFFFF0`	RAM empfängt Inhalt für `i`

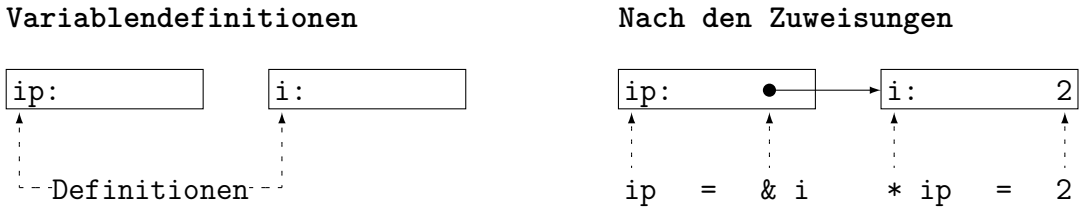
Schritte 1-3: Ok, die CPU soll eine Zuweisung ausführen; diese Maschineninstruktion steht bereits im Instruktionsregister. Durch die Maschineninstruktion weiß sie auch, dass in der folgenden RAM-Adresse (`0x086AD740`) auch das „erste“ Ziel steht. Also legt sie diese Adresse über ihren Program Counter auf den Adressbus. Der Arbeitsspeicher antwortet entsprechend mit dem Inhalt, also diesmal mit dem Wert `0xFFFFFFF8`. Diesen Wert leitet die CPU direkt in ihr Adress-Register, da sie dieses ja wie bereits besprochen auf den Adress-Bus legen kann. Ok, jetzt steht also der Wert `0xFFFFFFF8` im Adress-Register, der „zufällig“ die Adresse unserer Zeigervariablen `ip` ist.

Schritte 4-6: Jetzt wird wieder der gleiche Prozess wie in den drei vorherigen Schritten abgearbeitet. Die CPU legt ihr Adress-Register auf den Adress-Bus, in dem sich die Adresse unserer Zeigervariablen `ip` befindet. Der Arbeitsspeicher antwortet mit seinem Inhalt. An dieser Stelle befindet sich der Wert `0xFFFFFFF0`, der „zufällig“ die Adresse der eigentlichen Zielvariablen `i` ist. Da es sich bei dem eingehenden Wert wiederum um eine Adresse handelt, wird er wiederum in das Adress-Register geleitet. Jetzt steht also endlich die richtige Adresse, die der Variablen `i`, im Adress-Register.

Schritte 7-8: Nun kann die CPU endlich die eigentliche Zuweisung ausführen. Dazu legt sie das Adress-Register an den Adress-Bus und den Wert `2` über eines ihrer Daten-Register an den Daten-Bus, sodass der Arbeitsspeicher den Wert `2` an der Stelle `0xFFFFFFF0` ablegen kann, an der sich die Vairable `i` befindet. Jetzt ist alles gut!

Erklärungsversuch Bildliche C-Ebene:

Der dritte Erklärungsversuch betrachtet die Variablen auf der Ebene des C-Programms:

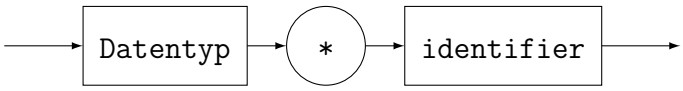


Man sieht deutlich, dass sich der Term `ip` immer auf die eigentliche Variable bezieht, dass durch die Zuweisung `ip = & i` ein „Zeiger“ von der Variablen `ip` zur Variablen `i` „einge-

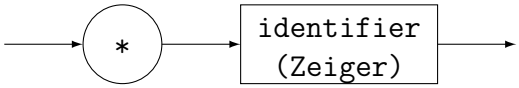
richtet“ wird und dass sich der Term `* ip` nicht auf die Variable `ip` (den „Zeigeranfang“) sondern auf das Ende des „Zeigers“ bezieht. Dabei muss man aber beachten, dass sich das C-Programm nicht dadurch „verzeigert“, dass man einen Zeiger auf das Blatt Papier malt, sondern dass der Zeiger durch die richtigen Anweisungen im C-Programm entsteht!

45.6 Syntax

Zeiger-Definition



Zeiger Dereferenzieren



In diesem Syntaxdiagramm kann „identifier“ bereits ein Zeiger-Typ sein. Damit lassen sich auch Zeiger auf Zeiger und Zeiger auf Zeiger auf Zeiger und so weiter definieren. Wo das alles hin führt, sehen wir in den Kapiteln 49 und 51. Beim Dereferenzieren, also dem Zugriff auf die entsprechenden Inhalte, muss man *nur* die richtige Zahl von Sternchen verwenden.

45.7 Interne Repräsentation

Bei allen Schwierigkeiten hat der Zeiger-Typ einen Vorteil: Egal worauf Zeiger so zeigen mögen, sie benötigen alle gleich viel Platz. Auf heutigen Rechnern sind dies in der Regel vier oder acht Bytes. Wie bekommt man das heraus? Klar, wieder mit der Compiler-Funktion `sizeof()`. Beispiel: `printf( "sizeof(int *)=%d Bytes\n", sizeof(int *) );`

45.8 Erlaubte Zeigerwerte

Da das Hantieren mit Zeigern des Öfteren zu Programmabstürzen führt, wird man natürlicherweise irgendwann unsicher, was für Werte man einem Zeiger zuweisen darf. Diese Frage lässt sich anhand folgender Regeln gut beantworten:

1. Eine Zeigervariable darf jeden beliebigen Wert annehmen. Da kann nichts passieren!
2. Auf den Speicher zugreifen (dereferenzieren des Zeigers) darf man nur, wenn der Zeigerwert einer gültigen Adresse im Arbeitsspeicher entspricht.
3. Der Zeigerwert `0` ist *sehr speziell*. Er symbolisiert einen *ungültigen* Zeigerwert, da man auf diese Speicherstelle nicht zugreifen darf. Entsprechend findet man in Programmen sehr häufig Abfragen der Art `if ( p != 0 )`, wobei `p` ein Zeiger ist.

45.9 Datentypen

Auch wenn wir hier nur ein erstes Beispiel besprochen haben, so sollten wir uns schon jetzt daran gewöhnen, dass es hier Typen gibt. Zur Übung zeigen wir hier noch eben ein paar Beispiele. Mit den folgenden Definitionen verbinden sich unten aufgeführte Datentypen:

```
1 int  * i_ptr;                // ein int-Zeiger
2 int  ** i_ptr_ptr;           // ein int-Zeiger-Zeiger
3 char * c_ptr, ** c_ptr_ptr; // beide Varianten in einer Zeile
4 double d;                    // ein double
```

Ausdruck	Typ	Kommentar
`i_ptr`	`int *`	Ein einfacher Pointer to (Zeiger auf) `int`
`* i_ptr`	`int`	Am Ende des Zeigers ist ein `int`
`i_ptr_ptr`	`int **`	Pointer to Pointer to `int`
`* i_ptr_ptr`	`int *`	Immer noch ein Pointer to `int`
`** i_ptr_ptr`	`int`	Am Ende des Zeiger-Zeigers ist ein `int`
`c_ptr`	`char *`	Pointer to `char`
`* c_ptr`	`char`	Am Ende des Zeigers ist ein `char`
`c_ptr_ptr`	`char **`	Pointer to Pointer to `char`
`* c_ptr_ptr`	`char *`	Immer noch ein Pointer to `char`
`** c_ptr_ptr`	`char`	Am Ende des Zeiger-Zeigers ist ein `char`
`d`	`double`	Ein `double`
`* d`	`---`	Nicht erlaubt, da `d` kein Zeiger ist
`& d`	`double *`	Könnten wir einer Variablen `double * dp` zuweisen

Kleine Eselsbrücke: In obiger Tabelle können wir folgendes erkennen: Für jede einzelne Variable gilt, dass die Summe der Sternchen je Zeile immer identisch mit der Zahl der Sternchen in der Definition ist: `i_ptr`: 1, `i_ptr_ptr`: 2, `c_ptr`: 1, `c_ptr_ptr`: 2, `d`: 0.

Jetzt wissen wir endlich, dass `argv` in `int main(int argc, char **argv)` ein Zeiger auf ein Zeiger auf ein `char` ist. Was das genau bedeutet, besprechen wir in Kapitel 51.

Kapitel 46

Arrays und Zeiger-Arithmetik

Zeiger sind zwar keine Zahlen im eigentlichen Sinne, aber dennoch kann man mit Zeigern auch rechnen. Das Rechnen mit ihnen ist sogar recht intuitiv. Allerdings gibt es ein paar kleine Unterschiede zwischen dem, was im C-Quelltext steht und was von der CPU tatsächlich gemacht wird. Der wesentliche Unterschied besteht darin, dass alle Größenangaben *intern* mit der Größe des dahinter liegenden Datentyps skaliert werden.

Die in diesem Kapitel besprochenen Adressberechnungen sind zwar prinzipiell unabhängig von Arrays, sind aber meist nur im Zusammenhang mit ihnen besonders sinnvoll. Um hier jegliches Missverständnis zu vermeiden, nochmals anders ausgedrückt: obiger Zusammenhang bezieht sich auf Adress*berechnungen*. Das Bilden von Adressen einzelner Variablen ist natürlich auch sinnvoll; aber in solchen Fällen ist das Anwenden der Adress-Arithmetik nur sehr vereinzelt sinnvoll.

46.1 Beispielkonfiguration

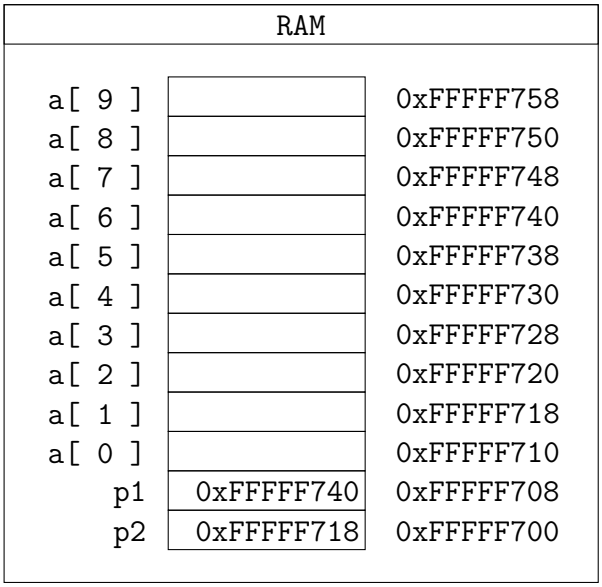
Im Laufe dieses Kapitels werden wir wiederholt auf ein Beispiel zurückgreifen, das wir in diesem Abschnitt wie folgt definieren.

Die Variablendefinition besteht aus einem Array `int a[10]` bestehend aus zehn Elementen vom Typ `int` sowie zwei Zeigern `int *p1, *p2` vom Typ Zeiger auf `int`, also vom Typ `int *`. Zur Erläuterung einiger Ausdrücke verwenden wir zusätzlich noch zwei weitere Zeiger `int *p3, *p4`, die wir aber nicht in das Speicherbild eintragen, da sie nur der Illustration dienen. Nach der Initialisierung `p1 = & a[ 6 ]` und `p2 = & a[ 1 ]` ergibt sich das auf der nächsten Seite gezeigte Speicherbild. Dieses Speicherbild geht davon aus, dass sowohl Variablen vom Typ `int` als auch Zeiger auf `int` acht Byte Speicherplatz benötigen. Ferner sind alle Adressen frei erfunden.

C-Programm

```
1  int a[ 10 ];
2  int * p1;
3  int * p2;
4
5  p1 = & a[ 6 ];
6  p2 = & a[ 1 ];
```

Auszug aus dem Arbeitsspeicher



46.2 Zeiger-Arithmetik

Addition/Subtraktion eines `int` Wertes: `p + i`

Wie oben im Vorspann bereits gesagt, werden alle Anweisungen im Quelltext intern mit der Speichergröße des zugrundeliegenden Datentyps skaliert. Für unser Beispiel von `int`-Zeigern und Arrays zuzüglich einer (kleine) `int`-Konstante `i` gilt folgender Zusammenhang:

C-Syntax	**Maschinenprogramm**
`p4 = p3 + i;`	`p4 ← p3 + i * sizeof(int);`
`p4 = p3 - i;`	`p4 ← p3 - i * sizeof(int);`

Auch wenn für `i` meist Konstanten genommen werden, so kann für `i` jeder beliebige, ganzzahlige Ausdruck substituiert werden. Für unser obiges Beispiel, würden sich folgende Zahlen ergeben:

C-Quelltext	Wertzuweisung im RAM	Äquivalente
`p3 = p1 + 1`	`p3 ← 0xFFFFF748`	`p3 = & a[6 + 1] = & a[ 7 ] = a + 7`
`p3 = p1 + 2`	`p3 ← 0xFFFFF750`	`p3 = & a[6 + 2] = & a[ 8 ] = a + 8`
`p3 = p1 - 2`	`p3 ← 0xFFFFF730`	`p3 = & a[6 - 2] = & a[ 4 ] = a + 4`

Da es sich hier um die Addition bzw. Subtraktion von Konstanten nebst einer Wertzuweisung handelt, können wir auch alle Kurzformen verwenden, die wir in Kapitel 43 besprochen haben: `p1 = p1 + 1`, `p1 += 1`, `p1++`, `++p1`, `p1--`, `--p1`.

Allgemeiner Fall: Im allgemeinen Fall, also beliebigem Basistyp, müssen wir obige Ausdrücke `sizeof(int)` durch `sizeof(* p1)` ersetzen, was uns aber nicht weiter belastet, da diese Umrechnungen nur intern vom Compiler durchgeführt werden.

Multiplikation/Division mit einer `int` Konstanten:
Dies ist nicht erlaubt.

Subtraktion zweier Zeiger: `p1 - p2`:
Oben im ersten Fall (Addition) hatten wir die Struktur `p4 = p3 + i`. Jetzt haben wir `i = p4 - p3`, was sofort klar macht, was das Ergebnis sein wird: Die Abarbeitung ist invers, die beiden Zeigerwerte werden von einander subtrahiert und das Ergebnis wird anschließend durch die Größe des Elementtyps dividiert. Mit `i` als eine Variable vom Typ `int` können wir den Effekt wie folgt veranschaulichen:

C-Syntax	**Maschinenprogramm**
`i = p4 - p3;`	`i ← (p4 - p3) / sizeof(int);`

Angewendet auf unser obiges Beispiel, würden sich folgende Zahlen ergeben:

C-Quelltext	Wertzuweisung im RAM	Äquivalente
`i = p1 - p2`	`i ← (0xFFFFF740-0xFFFFF718)/8 =  5`	`i = 6 - 1 =  5`
`i = p2 - p1`	`i ← (0xFFFFF718-0xFFFFF740)/8 = -5`	`i = 1 - 6 = -5`
`i = p1 - & a[2]`	`i ← (0xFFFFF740-0xFFFFF720)/8 =  4`	`i = 6 - 2 =  4`

Zusammenfassung: Die Subtraktion zweier Zeiger ermittelt die Zahl der Elemente, die sich zwischen ihnen befinden. Dabei *müssen* beide Zeiger vom selben Datentyp sein und für Programmieranfänger sinnvollerweise in das selbe Array zeigen.

Addition/Multiplikation/Division zweier Zeiger:
Dies ist nicht erlaubt.

Viel mehr ist zum Thema Zeiger-Arithmetik eigentlich nicht zu sagen. Interessant ist vielleicht noch, dass die Zeiger-Arithmetik gar nicht unbedingt das Setzen von Klammern voraussetzt. Ein kleines Beispiel sieht wie folgt aus:

```
& a[4] + 1 - & a[2] = (& a[4] + 1) - & a[2] = (& a[4] - & a[2]) + 1 = 3 .
```

Warum ist das so? Einfach mal am Beispiel auf einem Blatt Papier durchrechnen.

46.3 Kleines Beispielprogramm

Zur Illustration zeigt die nächste Seite ein kleines (eigentlich sinnloses) Beispielprogramm, in dem die Adress-Arithmetik an einigen Stellen beispielhaft angewendet wird. Anhand dieses Beispiels werden wir die wichtigsten Anweisungen im Detail besprechen.

```
 1 #include <stdio.h>
 2
 3 #define SIZE    10
 4
 5 int main( int argc, char **argv )
 6     {
 7         int a[ SIZE ], i, * ip;
 8         for( ip = & a[ 0 ]; ip < & a[ 0 ] + SIZE; ip++ )
 9             *ip = ip - & a[ 0 ];
10         for( i = 0; i < SIZE; i++ )
11             printf( "a[ %d ]= %d\n", i, a[ i ] );
12     }
```

Zeile 7:

Hier sind alle Definitionen zu finden. Sie bestehen aus einem Array `a` mit zehn Elementen vom Typ `int`, einer „generischen“ (Lauf-) Variablen `i` und einem Zeiger `ip` auf ein `int`.

Zeilen 8 und 9:

In dieser `for`-Schleife erhalten alle zehn Elemente `a[ i ] = i` ihren Wert, der zufälligerweise ihrem Index entspricht. Hier hätten wir auch etwas anderes programmieren können, doch so ist es noch einigermaßen übersichtlich hat aber dennoch etwas zu bieten.

Zeile 8:

Diese Zeile bildet den Schleifenkopf. In dieser Schleife wird die Variable `ip` der Reihe nach auf die zehn Elemente `ip = & a[ 0 ]`, `ip = & a[ 1 ]`, ... `ip = & a[ 9 ]` gesetzt. Dazu wird anfänglich der Zeiger auf das erste Element `a[ 0 ]` (korrekt: die Zeigervariable `ip` bekommt als Wert die Adresse des ersten Elements) gesetzt. Am Ende jedes Schleifendurchlaufs wird der Zeiger mittels `ip++` jeweils auf das nächste Element „weitergeschaltet.“ Der Schleifenrumpf wird solange ausgeführt, wie der Zeiger `ip` noch innerhalb des Arrays ist, denn `& a[ 0 ] + SIZE` wäre die Adresse von `a[ 10 ]`, das direkt hinter dem letzten Element `a[ 9 ]` läge, wenn es denn existierte.

Zeile 9:

Der Ausdruck `ip - & a[ 0 ]` bestimmt, wie viele Elemente sich zwischen der aktuellen Zeigerposition und dem Anfang des Arrays befinden. Aufgrund des Schleifenkopfes sind dies der Reihe nach die Werte `0`, `1`, ..., `9`. Dieser Wert wird jeweils in diejenige Speicherzelle geschrieben, auf die durch den Inhalt von `ip` verwiesen wird.

Zeilen 10 und 11:

In dieser Schleife werden die Werte der Elemente wie gewohnt ausgegeben.

Noch irgendwelche Unklarheiten? Dann einfach mal eine Papier-und-Bleistift-Simulation anhand des Bildchens aus dem ersten Abschnitt dieses Kapitels machen.

46.4 Was ist ein Array, was dessen Name?

Nun schauen wir nochmals auf den Zusammenhang von Arrays und Zeigern, was nach den vorherigen Abschnitten zum Thema Adress-Arithmetik eigentlich recht einfach sein sollte. Die folgenden Erläuterungen beziehen sich auf unser Standardbeispiel `int a[ 10 ]`.

1. Durch diese Definition werden im Arbeitsspeicher zehn Elemente angelegt, die jeweils mittels `a[ 0 ]`, `a[ 1 ]`, ..., `a[ 9 ]` angesprochen werden können.
2. Der Name des Arrays, in diesem Fall `a`, ist eine *Konstante*, die die Adresse des ersten Elements angibt. Es gilt also (ganz allgemein): `a == & a[ 0 ]`.
3. Wenn der Name des Arrays der Adresse seines ersten Elements entspricht, dann können wir die Adresse der einzelnen Elemente auch wie folgt berechnen: `a + i == & a[ i ]`.
4. In obigem Programm könnten wir die erste `for`-Schleife (Zeilen 8 und 9) wie folgt vereinfachen:

```
8          for( ip = a; ip < a + SIZE; ip++ )
9             *ip = ip - a;
```

5. Da der Name des Arrays, in unserem Falle `a`, eine Konstante ist, kann ihr Wert auch nicht verändert werden. Alle Versuche der Art: `a += 3`, `a = b` etc. sind nicht erlaubt.
6. Da der Array-Name einem Zeiger entspricht, wandelt der Compiler *alle* Elementzugriffe wie folgt um: `a[ i ]` $\Rightarrow$ `*(a + i)`. Entsprechend wird aus einer einfachen Zuweisung `a[ 0 ] = a[ 1 ]` die „Zeigerei“: `*a = *(a + 1)`.
7. Auch wenn es nicht ganz korrekt ist und sich die Lehrenden nicht ganz einig sind, so hilft es vielen, wenn sie bei `a[]` an das gesamte Array denken und bei `a` nur an die Anfangsadresse des Arrays.

46.5 Ausdrücke und Kurzformen

Um folgendes noch explizit zu machen: Alle in diesem Kapitel besprochenen Ausdrücke (Zeiger-Arithmetik) sind schon vom Namen her Ausdrücke. Natürlich sind auch Zeiger und Adressen Ausdrücke.

Innerhalb von Ausdrücken können wir verschiedene Operatoren miteinander kombinieren. Dies betrifft auch das Sternchen (dereferenzieren) und die beiden Pre-/Post- Decrement/Increment Operatoren `++` und `--`, die stärker binden als das Sternchen. Diese Kombinationsmöglichkeit erzeugt fast allen Anfägern und vielen Fortgeschrittenen richtige Kopfschmerzen. Um dem vorzubeugen, präsentieren wir in diesem Abschnitt ein paar Beispiele. Diese gehen davon aus, dass wir vor jedem Ausdruck immer folgende Konfiguration haben:

```
1 char c[ 3 ] = { 'A', 'D', 'F' };
2 char *p = c + 1;                  // p == & c[ 1 ]
3 int  i;
```

Ausdruck	geklammert	Effekt auf `c`, `p` und `i`
`i = *p++`	`i = *(p++)`	`i == 'D', c == { 'A', 'D', 'F' }, p == & c[ 2 ]`
`i = *++p`	`i = *(++p)`	`i == 'F', c == { 'A', 'D', 'F' }, p == & c[ 2 ]`
`i = *p--`	`i = *(p--)`	`i == 'D', c == { 'A', 'D', 'F' }, p == & c[ 0 ]`
`i = *--p`	`i = *(--p)`	`i == 'A', c == { 'A', 'D', 'F' }, p == & c[ 0 ]`
`i = (*p)++`		`i == 'D', c == { 'A', 'E', 'F' }, p == & c[ 1 ]`
`i = ++(*p)`		`i == 'E', c == { 'A', 'E', 'F' }, p == & c[ 1 ]`
`i = (*p)--`		`i == 'D', c == { 'A', 'C', 'F' }, p == & c[ 1 ]`
`i = --(*p)`		`i == 'C', c == { 'A', 'C', 'F' }, p == & c[ 1 ]`

Kapitel 50 präsentiert einige kleine Beispielprogramme, die solche Kurzformen verwenden.

Kapitel 47

Funktionen mit Arrays und Zeigern

In Kapitel 44 haben wir Euch gezeigt, wie man seine eigenen Funktionen implementieren kann. Mit ein klein wenig Übung sind Funktionen eine schöne Sache. In Kapitel 44 haben wir ebenfalls sehr eindringlich darauf hingewiesen, dass Funktionen *immer* auf lokalen Kopien der aktuellen Parameter arbeiten. Eine unmittelbare Konsequenz dieses Ansatzes ist, dass alle Änderungen auf der Seite der formalen Parameter (also innerhalb der Funktion) keine Wirkung bezüglich der aktuellen Parameter (also außerhalb der Funktion) haben.

Im Sinne des Software Engineerings ist diese Funktionalität aber manchmal zu eingeschränkt, denn manchmal möchte man einfach einen Seiteneffekt aus einem Unterprogramm heraus haben. Dies widerstrebt aber dem Grundgedanken einer Funktion, die nichts verändert und nur einen Rückgabewert liefert. Als Ausweg bieten andere Programmiersprachen das Konzept der *Procedure*. Diese sind fast das gleiche wie Funktionen, nur dass sie keinen Rückgabewert haben und dass sich Änderungen der formalen Parameter auf die aktuellen Parameter auswirken können. Parameter, bei denen sich Änderungen der formalen Parameter auf die aktuellen Parameter auswirken, nennt man auch Variablenparameter, wofür viele Programmiersprachen extra einen zweiten Parameterübergabemechanismus[1]genannt Call-by-Reference besitzen; in Pascal wird diesen formalen Parametern einfach das Schlüsselwort `var` vorangestellt.

Call-by-Reference gibt es aber in C definitiv *nicht*, auch wenn dies immer wieder behauptet wird. In der Programmiersprache C gibt es definitiv nur den Parameterübergabemechanismus Call-by-Value! Aber die gute Nachricht ist, dass wir dies mit den bereits gelernten Mitteln, insbesondere den Zeigern, recht einfach selbst implementieren können. In diesem Kapitel schauen wir uns an, wie Arrays an Funktionen übergeben werden und was man mit Formalen Parametern machen kann, die vom Typ Zeiger sind. Für das weitere Verständnis sind neben Kapitel 44 vor allem die Kapitel 45 und 46 Voraussetzung.

[1]In einigen Programmiersprachen gibt es noch weitere Paraemterübergabemechanismen, auf die wir hier aber nicht weiter eingehen.

47.1 Wiederholung

Als Einstieg sei nochmals an die folgenden Dinge erinnert:

1. Zeiger definiert man mittels des Sternchens `*`, beispielsweise `int * ip`. Bei der Verwendung eines Zeigers muss man unerscheiden, ob man die Zeigervariable anspricht, oder die Zeigervariable dereferenziert (also entlang des Zeigers geht und die Speicherstelle anspricht, die in der Zeigervariablen als Wert steht). Beispielsweise muss man zwischen den beiden Anweisungen `ip = & i` und `* ip = 3` ganz genau unterscheiden. Wer hier unsicher ist, sollte unbedingt nochmals die Abschnitte 45.4 und 45.5 durcharbeiten.
2. Zu jeder Funktion gehört ein Stack Frame, der die formalen Parameter, den vorherigen Wert des Program Counters und die lokalen Variablen enthält. Bei jedem Funktionsaufruf wird dieser Stack Frame auf dem Stack angelegt und am Ende der Funktion auch wieder vollständig entfernt. Im Zweifelsfalle einfach nochmals Kapitel 44 durcharbeiten.

Wichtiger Hinweis: Ohne das Verständnis dieser Voraussetzungen ist das Weiterlesen dieses Kapitel nicht besonders sinnvoll ...

47.2 Zeiger als Parameter einer Funktion

Springen wir ins kalte Wasser und schauen uns folgendes Programm an:

```
1  #include <stdio.h>
2
3  int swap( int * ip1, int  * ip2 )
4      {
5         int h;
6         h = * ip1; * ip1 = * ip2; * ip2 = h;
7         return 1;
8      }
9
10 int main( int argc, char **argv )
11     {
12        int a = 1, b = 2;
13        swap( & a, & b );
14        printf( "a= %d b= %d\n", a, b );
15     }
```

Wie immer besprechen wir im folgenden das Programm in allen Einzelheiten:

Zeile 3-8:
In diesen sechs Zeilen wird die Funktion `swap()` implementiert. Diese Funktion hat

zwei Parameter `ip1` und `ip2`, die beide vom Typ `int *` sind. Diese beiden Parameter sind also Zeiger auf `int`.

Zeile 13:
Hier wird die Funktion `swap()` mit den beiden Parametern `& a` und `& b` aufgerufen. Hier werden also nicht die Werte der beiden Variablen `a` und `b` sondern deren Adressen an die Funktion `swap()` übergeben. Durch diesen „Trick“ weiß nun die Funktion, *wo* sich die beiden Variablen `a` und `b` im Arbeitsspeicher befinden. Zur Erinnerung: Die aktuellen Parameter sind nicht die Werte der beiden Variablen `a` und `b` sondern deren Adressen.

Zeilen 3 und 13:
Die aktuellen Parameter `& a` und `& b` haben den selben Datentyp `int *` (Zeiger auf `int`) wie die beiden formalen Parameter `ip1` und `ip2`. Bis hierher ist also schon mal alles gut.

Zeile 6:
Hier kommt zum tragen, was wir in obiger Wiederholung nochmals angesprochen haben: in dieser Zeile verändern wir nicht die Zeiger (das hätte auch gar keine Wirkung) sondern diejenigen Speicheradressen, auf die durch die beiden Zeiger-Variablen `ip1` und `ip2` verwiesen (referenziert) wird.

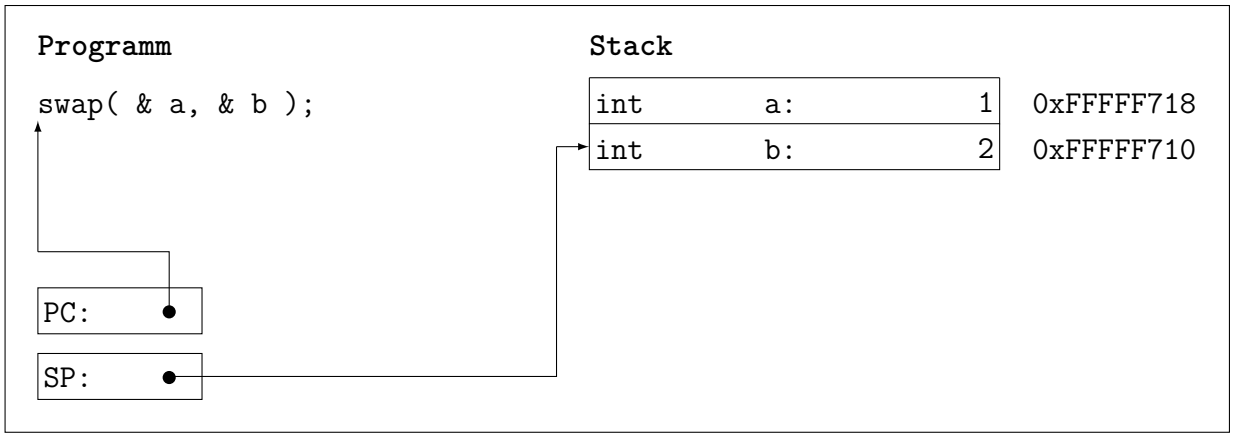
Parameterübergabe Call-by-Value: Um es hier nochmals ganz deutlich und ohne jegliche Missverständnisse zu sagen: Auch wenn in diesem Beispiel Zeiger, also Referenzen auf andere Objekte, übergeben werden, so handelt es sich dennoch um Call-by-Value, denn die Adressen (Werte) werden einfach in die formalen Parameter hinein kopiert. Sollten diese lokalen Zeiger (Adressen) innerhalb der aufgerufenen Funktion verändert werden, so hat dies *keinen* Einfluss auf irgendwelche anderen Variablen, die außerhalb dieser Funktion sind. Sollten wir aber die Zeiger verändern *und* dann auch noch dereferenzieren, könnte ziemlicher Unsinn entstehen. Da wir hier an einer Universität sind, diskutieren wir das Thema Call-by-Reference nochmals kurz am Ende dieses Kapitels.

Funktionsaufruf und Abarbeitung über den Stack (Frame): Im Folgenden zeigen wir noch kurz, wie obiges Beispielprogramm über den Stack abgewickelt wird. Die Funktion `swap()` hat folgenden Stack Frame:

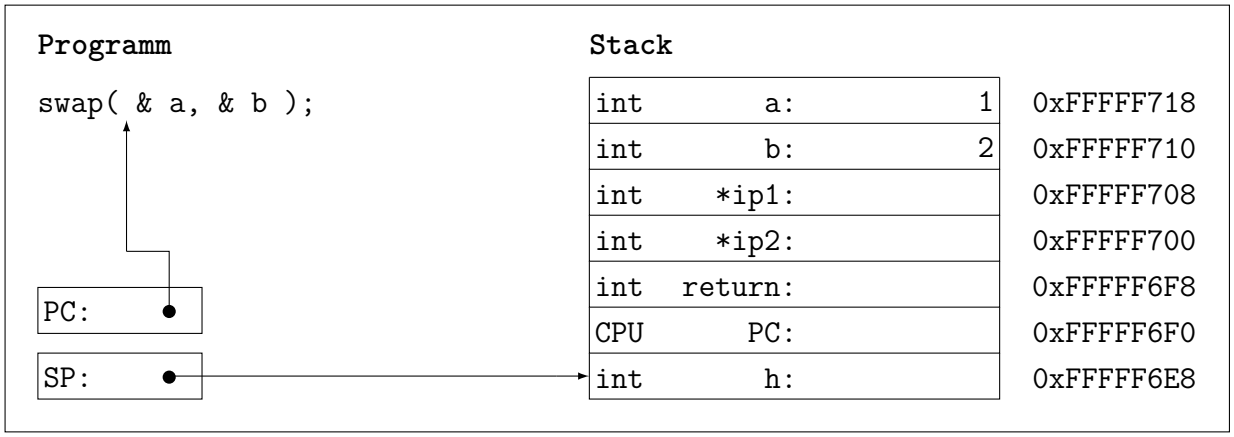
Stack Frame: Funktion: swap()

`int *ip1:`
`int *ip2:`
`int return:`
`CPU PC:`
`int h:`

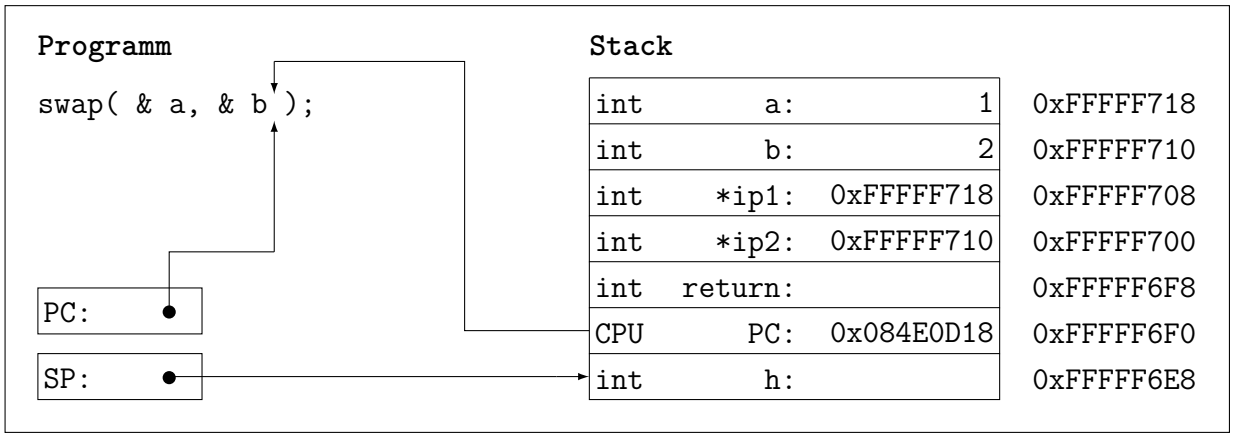
Die folgende Grafik zeigt wieder den Stack unmittelbar vor dem Aufruf der Funktion `swap()`. Im Vergleich zu Kapitel 44 sind aus Platzgründen die obersten Einträge des Stacks weggelassen, da sie hier nichts beitragen würden.



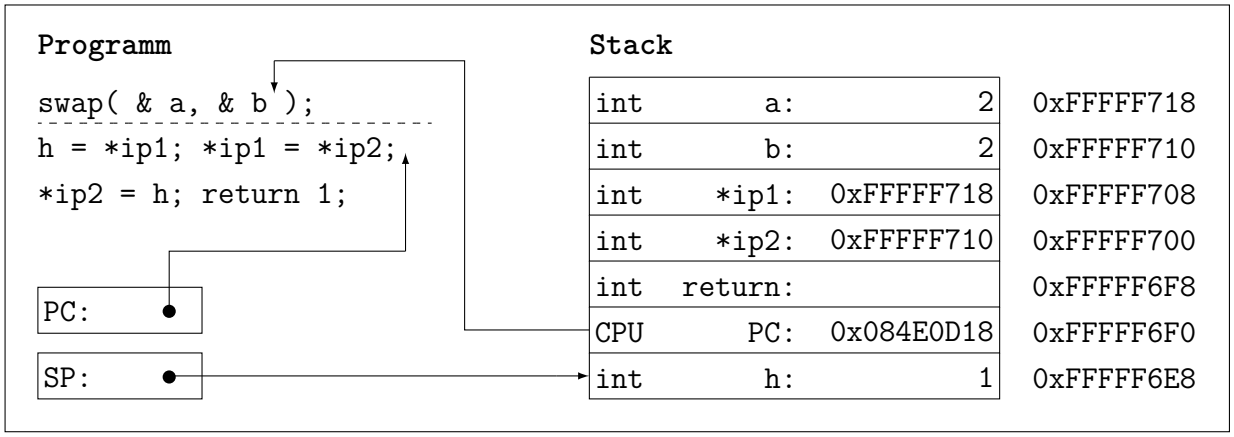
Die erste Aktion beim Funktionsaufruf ist das Anlegen des neuen Stack Frames. Dies geschieht in gewohnter Art und Weise:

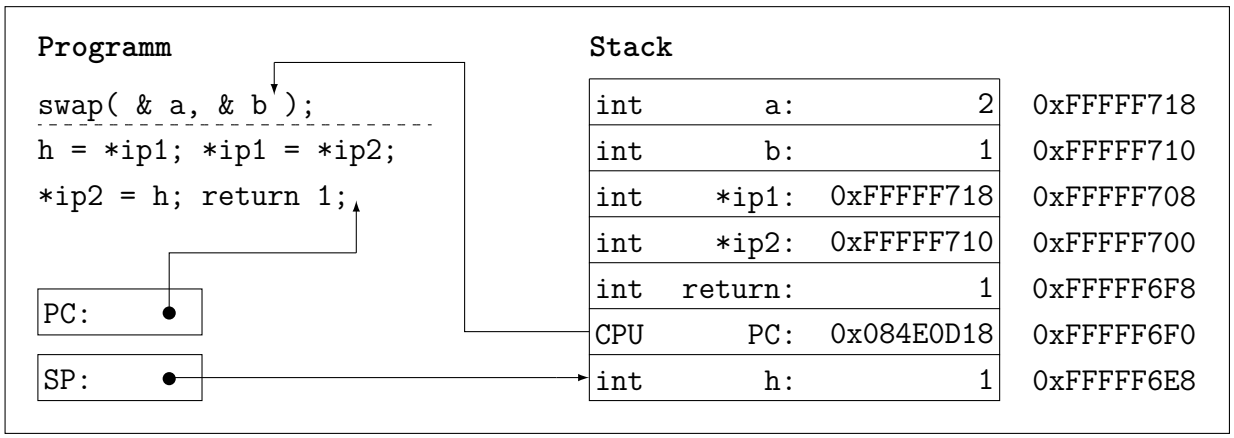


Als nächstes werden die beiden Adressen `& a` und `& b` in die formalen Parameter kopiert, sodass der Funktion `swap()` jeweils eine Referenz auf die beiden aktuellen Parameter hat:

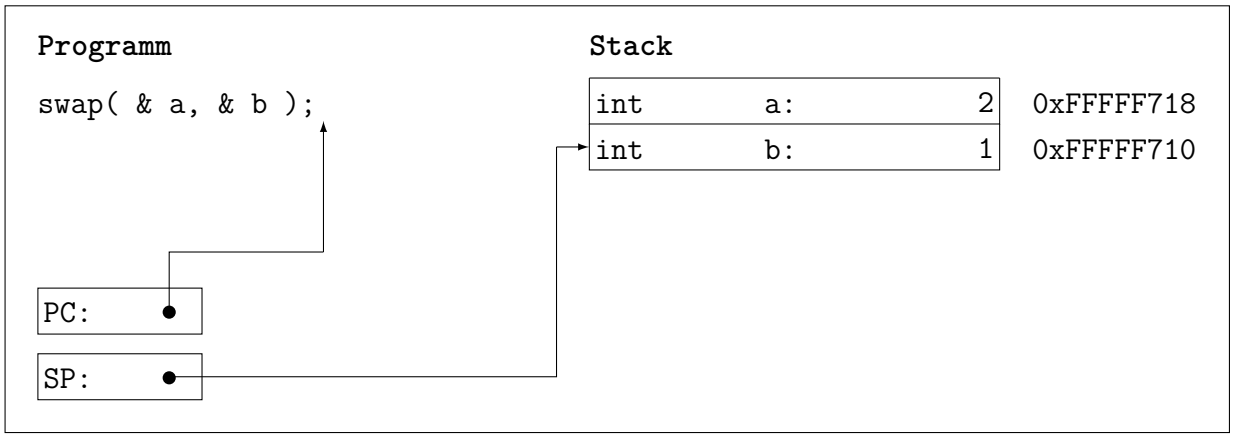


Die nächsten beiden Grafiken zeigen die Abarbeitung der Funktion `swap()` bis unmittelbar vor dem Ende der `return`-Anweisung:





Als letztes wird nach dem Rücksprung zur alten Stelle im „Hauptprogramm“ der Stack Frame der Funktion `swap()` wieder entfernt. Und ganz nebenbei haben die beiden Variablen `a` und `b` ihre Werte getauscht.



Und am Ende sieht der Stack wieder aus wie vorher ...

47.3 Arrays als Parameter einer Funktion

Bevor wir jetzt besprechen, was das letzte Programm aus Kapitel 44 macht, sei nochmals an den Inhalt von Kapitel 46 erinnert (im Zweifelsfalle nochmals ansehen):

1. Der Name des Arrays repräsentiert die Adresse des ersten Elementes dieses Arrays. Beispiel: `int a[10]` ⇒ `Typ(a): int *`.
2. Zugriffe auf einzelne Array-Elemente werden in ein Dereferenzieren eines Zeigers umgewandelt: Beispiel: `a[4] = -2` ⇒ `*(a + 4) = -2`.

Hier zur Wiederholung nochmals das Beispiel aus Kapitel 44:

```
 1  #define SIZE    10
 2
 3  int set_to_1( int a[], int size )
 4      {
 5          int i;
 6          for( i = 0; i < size; i++ )
 7              a[ i ] = 1;
 8          return 1;
 9      }
10
11  int main( int argc, char **argv )
12      {
13          int arr[ SIZE ];
14          set_to_1( arr, SIZE );
15      }
```

Nach dem bisher Gelernten sollte eigentlich ziemlich klar sein, was hier passiert:

Zeile 14: `set_to_1( arr, SIZE );`
Klar, `arr` repräsentiert die Adresse des ersten Elementes des Arrays `arr`. Entsprechend wird hier eine Adresse übergeben, die den Datentyp `int *` hat.

Zeile 3: `int set_to_1( int a[], int size )`
Nach dem aktuellen Sprachstandard hätte hier auch stehen können: `int set_to_1( int *a, int size )`. Dies bedeutet, dass wir *innerhalb* der Funktion `set_to_1()` einen Zeiger `a` haben, der „zufällig“ auf das Array `arr` aus dem Hauptprogramm zeigt. Alle weiteren Zugriffe auf dieses Array erfolgen wie in Kapitel 46 beschrieben.

Zeile 7: `a[ i ] = 1;`
Wir erinnern uns nochmals, dass diese Zeile identisch ist mit `*(a + i) = 1`.

Zeilen 3 und 14:
Da die Funktion `set_to_1()` nur eine Adresse mitten in den Arbeitsspeicher bekommt, kann sie erstens nicht wissen, dass es sich um das Array `arr` handelt, und

kann zweitens daher auch nicht dessen Größe bestimmen. Deshalb *muss* der Funktion die Größe des Arrays übergeben werden.

Nach dem hier und in Kapitel 46 Gelernten hätten wir die Funktion `set_to_1()` auch wie folgt schreiben können (beide Varianten wären funktional identisch mit dem Originalbeispiel):

Alternative 1:

```
1  int set_to_1( int *a, int size )
2      {
3          int i;
4          for( i = 0; i < size; i++ )
5              *(a + i) = 1;
6          return 1;
7      }
```

Alternative 2:

```
1  int set_to_1( int *a, int size )
2      {
3          int *end_p = a + size;  // end_p zeigt hinter das array
4          for( ; a < end_p; a++ )
5              *a = 1;     // a wird ja immer eins weiter geschaltet
6          return 1;
7      }
```

47.4 Arrays und Funktionen: Variationen

Der von der Programmiersprache C gewählte Weg bei der Übergabe von Arrays an Funktionen eröffnet auch eine Reihe von interessanten Möglichkeiten, die man in anderen Programmiersprachen nicht hat. Für obige Beispielfunktion `set_to_1()` zeigt die folgende Tabelle, welche der Array-Elemente auf den Wert `1` gesetzt werden:

Aufruf	Betroffene Elemente
`set_to_1( arr    , 10 )`	`arr[ 0 ] .. arr[ 9 ]`
`set_to_1( arr    ,  4 )`	`arr[ 0 ] .. arr[ 3 ]`
`set_to_1( arr + 1,  7 )`	`arr[ 1 ] .. arr[ 7 ]`
`set_to_1( arr + 4,  1 )`	`arr[ 4 ]`
`set_to_1( arr + 2, 10 )`	fehlerhaft, da außerhalb der Array-Grenzen
`set_to_1( arr - 1,  2 )`	fehlerhaft, da außerhalb der Array-Grenzen

47.5 Array-Definition vs. Arrays als Parameter

Nun greifen wir nochmals ein Thema auf, das wir bereits in Kapitel 46.4 angesprochen haben: Sind Arrays nun Zeiger oder nicht? Eine pauschale Antwort wäre ein eindeutiges „jein“! Die richtige Antwort hängt davon ab, wo im Programm man sich gerade befindet:

Definition:
Dort, wo das Array deklariert wird, kann man unter `array[ n ]` das gesamte Array mit `n` Elementen verstehen. D.h., durch diese Definition hat man genau `n` Elemente, denen man eigene Werte zuweisen kann. Gleichzeitig ist `array` eine Konstante, die die Adresse des ersten Elementes dieses Arrays repräsentiert. Da eine Konstante ein Objekte ist, das konstant ist (sagt ja bereits der Name), kann diese Konstante durch nichts verändert werden.

Als Parameter einer Funktion:
Wenn man ein Array an eine Funktion übergibt, wird nur der Name des Arrays angegeben. Dieser Name ist, wie eben gesagt, eine Konstante, die eine Adresse repräsentiert. Entsprechend handelt es sich um den Typ „Zeiger auf“. Demzufolge wird der entsprechende Parameter im Funktionskopf auch als Zeiger deklariert. Somit hat man jetzt *innerhalb* der Funktion *einen* Zeiger, der auf die entsprechenden Elemente des übergebenen Arrays zeigt.

47.6 Hintergrunddiskussion: Call-by-Reference

Weiter oben haben wir versprochen, für die Interessierten noch kurz auf den Begriff Call-by-Reference einzugehen. Da es diesen Parameterübergabemechanismus in C nicht gibt, müssen wir zur Erklärung eine andere Programmiersprache wie beispielsweise Pascal heranziehen.

```
1 procedure p( var x: integer )
2 begin
3     x := 1;
4 end;
```

Würden wir jetzt diese Procedure `p( a )` aufrufen, wobei `a` eine Variable vom Typ `integer` ist, würde der Compiler daraus von sich aus ganz alleine entsprechende Adressen generieren und die Zeiger dereferenzieren. Mit anderen Worten, aus dem Aufruf würde durch den Compiler `p( & a )` (in C-Syntax) und innerhalb der Procedure würde aus Zeile 2 die Anweisung `* x = 1;` (wieder in C-Syntax) gemacht werden. Und retrospektivisch betrachtet ist dies in C auch so, nur müssen wir es selbst per Hand programmieren.

Kapitel 48

Rekursion

Rekursion liegt vor, wenn sich eine Funktion selbst aufruft. Wenn man die Arbeitsweise einer Funktion, insbesondere die Verwendung der Stack Frames, verstanden hat, ist Rekursion eigentlich ein super einfaches Thema. Aber leider bekommen viele Studenten bei diesem Thema Hautausschlag. Kurzgefasst: Es spricht überhaupt nichts dagegen, wenn sich eine Funktion selbst aufruft. Nach Bearbeiten des Funktionskopfes weiß der Compiler, wie die Funktion, insbesondere ihr Typ und ihre Parameter, aussieht. Also kann sie sich im Funktionsrumpf selbst aufrufen. Eine Schwierigkeit: Wenn sich eine Funktion *immer* wieder selbst aufruft, gibt es eine Endlosrekursion, die häufig fälschlicherweise Endlosschleife genannt wird. Aber egal, wie wir es nennen, es dauert unendlich lange. Um dies zu vermeiden, benötigt man im Funktionsrumpf ein Abbruchkriterium und alles wird gut.

48.1 Fakultät, ein klassisches Beispiel

In der Mathematik ist *eine* Definition der Fakultät $n!$ einer Zahl n wie folgt:

$$n! = 1 \times 2 \times \cdots \times n \quad \text{für } n \geq 0$$

Wer aber mal in ein schlaues Mathematikbuch schaut, findet auch folgende Defintion:

$$n! = \begin{cases} 1 & \text{für } n \leq 1 \\ n \times (n-1)! & \text{für } n > 1 \end{cases}$$

Und dies ist eine wunderschöne Rekursion. Die Rekursion besteht darin, dass die Fakultät sich in der unteren Zeile über sich selbst definiert. Wichtig ist aber auch die erste Zeile, denn sie terminiert die Rekursion für Argumente kleiner/gleich 1. Und dies können wir direkt, ohne jegliche Komplikation in C umsetzen:

```
1  int fakultaet( int n )
2      {
3         if ( n <= 1 )
4              return 1;
5         else return n * fakultaet(n - 1);
6      }
```

Das ist auch schon alles! Es sei noch angemerkt, dass wir ohne weitere Probleme auch schreiben könnten: `if ( n > 1 ) return n * fakultaet(n - 1) else return 1;` Es hätte absolut keinen Unterschied gemacht!

48.2 Abarbeitung der Rekursion

Eigentlich sollte klar sein, wie obige Rekursion abgearbeitet wird: bei jedem Aufruf wird ein neuer Stack Frame angelegt und die Funktion erneut gestartet. Erst wenn die neu gestartete Funktion (Instanz) zu Ende ist, macht die alte weiter.

Für diejenigen, die damit so ihre Schwierigkeiten haben, haben wir das ganze einmal für $n = 3$ aufgemalt, was viel Arbeit war. Für die folgende Diskussion haben wir die Implementierung ein wenig geändert. Funktional ist sie identisch, aber sie bietet uns ein paar mehr Zeilen, sodass wir die Abarbeitung ein wenig besser nachvollziehen können.

```
1  int fakultaet( int n )
2      {
3         int tmp;
4         if ( n <= 1 )
5              return 1
6                 ;
7         else {
8                tmp = fakultaet(n - 1)
9                      ;
10               return n * tmp
11                     ;
12             }
13     }
14
15 int main( int argc, char **argv )
16     {
17        int result;
18        result = fakultaet( 3 )
19                  ;
20        // hier steht das Ergebnis fest
21     }
```

Zunächst schauen wir uns den Stack Frame der Funktion `int fakultaet(int n)` an:

Stack Frame: Funktion: int fakultaet(int n)

int n:	
int return:	
CPU PC:	
int tmp:	

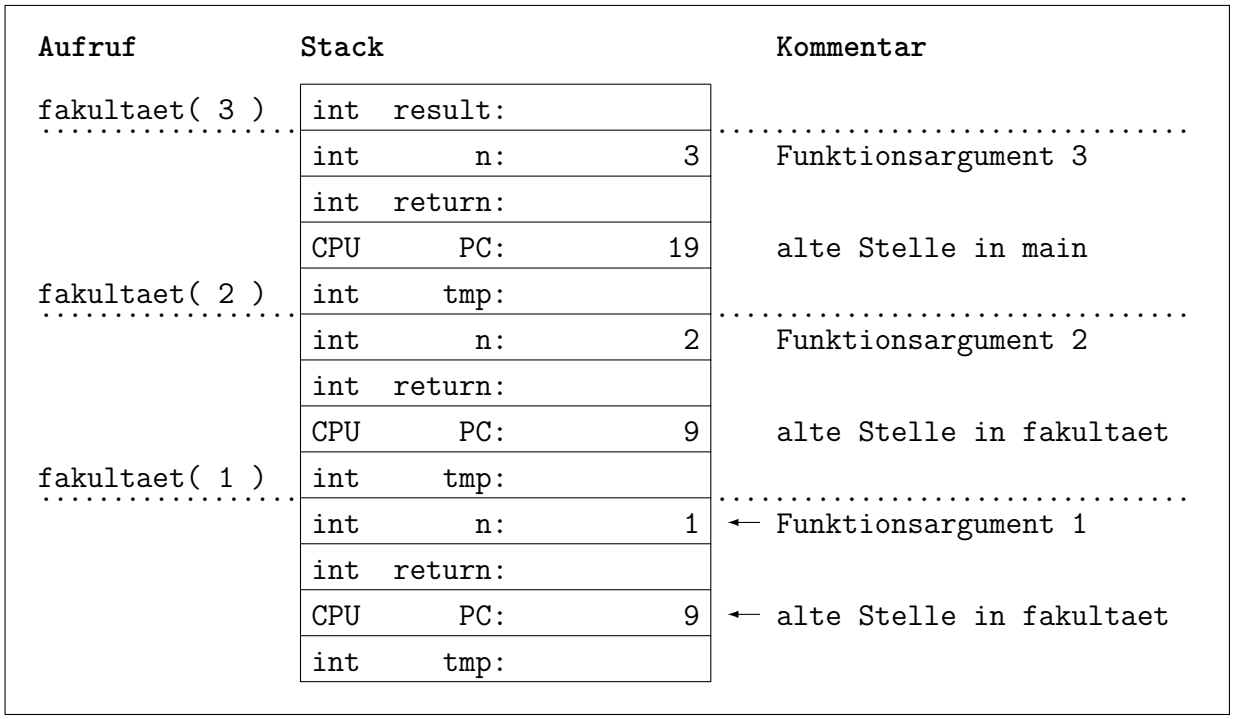
In den folgenden Bildern sehen wir von den konkreten, aber dennoch unhandlichen Speicheradressen ab und verwenden die Zeilennummern als Werte für den PC (Program Counter). Das erste Bild zeigt den Stack unmittelbar nach dem Aufruf von `fakultaet( 3 )`. Im Hauptprogramm müsste als nächstes die Zuweisung abgeschlossen werden, sodass sich der Stack Frame den Wert `PC = 19` merkt. Durch die Parameterübergabe gilt `n = 3`.

Aufruf	Stack		Kommentar
fakultaet(3)	int result:		
	int n:	3	← Funktionsargument 3
	int return:		
	CPU PC:	19	← alte Stelle in main
	int tmp:		

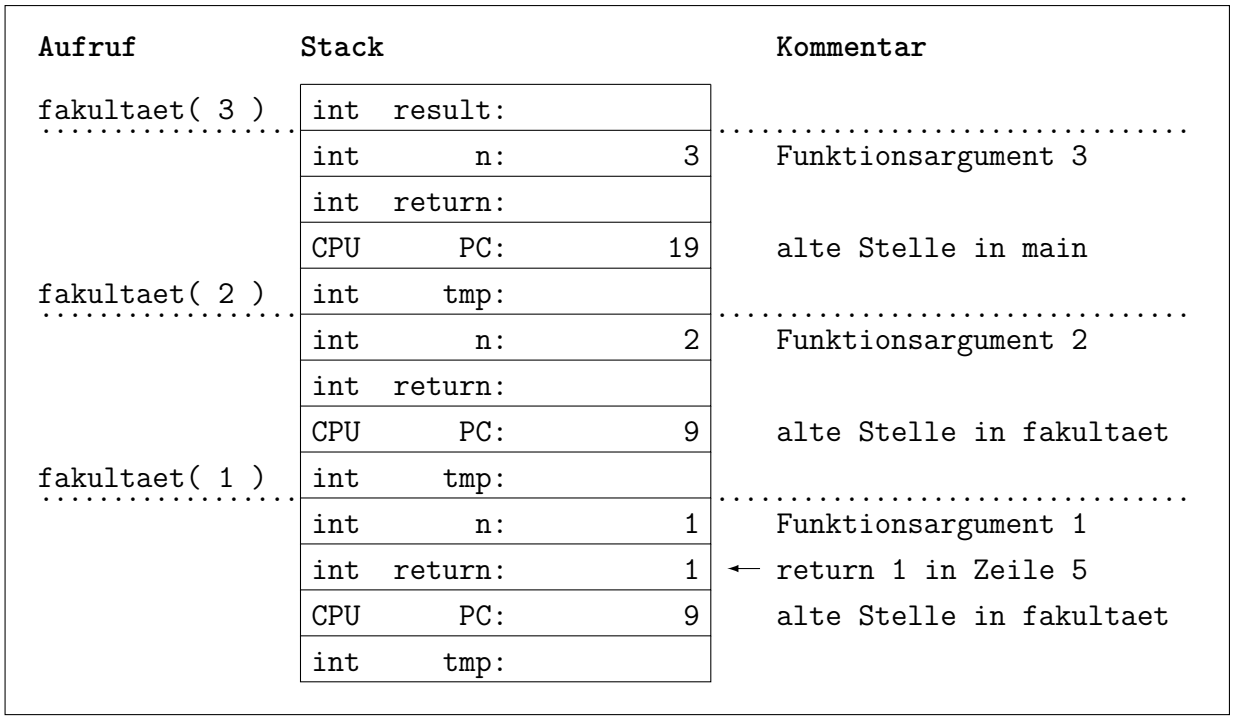
Da der Wert `3` deutlich größer als `1` ist, wird als nächstes der `else`-Teil abgearbeitet. Entsprechend wird die Funktion `fakultaet( 2 )` mit dem Argument `2` aufgerufen. *Nach* dem Aufruf muss der Rückgabewert noch der Variablen `tmp` zugewiesen werden, weshalb der „gemerkte“ PC den Wert `9` bekommt:

Aufruf	Stack		Kommentar
fakultaet(3)	int result:		
	int n:	3	Funktionsargument 3
	int return:		
	CPU PC:	19	alte Stelle in main
fakultaet(2)	int tmp:		
	int n:	2	← Funktionsargument 2
	int return:		
	CPU PC:	9	← alte Stelle in fakultaet
	int tmp:		

Das meiste ist wie gehabt. Also wird wieder die Funktion `fakultaet( 1 )` aufgerufen, aber diesmal mit dem Argument `1`:



Da nun das Argument `n = 1` nicht mehr größer als `1` ist, werden diesmal die Zeilen 5 und 6 abgearbeitet. Nach Ausführen der Zeile 5 sieht der Arbeitsspeicher wie folgt aus:



Mit Abarbeiten von Zeile 6, dem Semikolon, wird das `return` und damit auch der Funktionsaufruf `fakultaet(1)` abgeschlossen. Damit wird der Stack Frame vom Stack entfernt

und die CPU kann die vorher verlassene Zeile 9 abarbeiten, was dazu führt, dass der Rückgabewert an die Variable `tmp` übergeben wird. Anschließend wird `n * tmp` ausgerechnet:

Aufruf	Stack			Kommentar
`fakultaet( 3 )`	`int`	`result:`		
	`int`	`n:`	`3`	`Funktionsargument 3`
	`int`	`return:`		
	`CPU`	`PC:`	`19`	`alte Stelle in main`
`fakultaet( 2 )`	`int`	`tmp:`		
	`int`	`n:`	`2`	`Funktionsargument 2`
	`int`	`return:`	`2`	← `tmp * n in Zeile 10`
	`CPU`	`PC:`	`9`	`alte Stelle in fakultaet`
	`int`	`tmp:`	`1`	← `tmp=.... in Zeile 8`

Jetzt passiert wieder das gleiche. Durch Abarbeiten von Zeile 11 wird der Funktionsaufruf `fakultaet( 2 )` beendet und die CPU kann mit Zeile 9 des vorherigen Aufrufs weitermachen. Entsprechend geht der Rückgabewert wieder an die Variablen `tmp`. Anschließend wird wieder der neue Rückgabewert ausgerechnet, was zu folgender Situation führt:

Aufruf	Stack			Kommentar
`fakultaet( 3 )`	`int`	`result:`		
	`int`	`n:`	`3`	`Funktionsargument 3`
	`int`	`return:`	`6`	← `tmp * n in Zeile 10`
	`CPU`	`PC:`	`19`	`alte Stelle in main`
	`int`	`tmp:`	`2`	← `tmp=.... in Zeile 8`

Hier passiert wieder genau das gleiche. Mit Zeile 11 ist dann auch der Funktionsaufruf `fakultaet( 3 )` zu Ende. Durch Abarbeiten der Zeile 19 wird der Rückgabewert an die Variable `result` übergeben, womit der `PC` dann den Wert 20 erhält. Und wir sind fertig!

Aufruf	Stack			Kommentar
	`int`	`result:`	`6`	← `Zuweisung Zeile 19`

Das Ergebnis von `fakultaet( 3 )` steht endlich in der richtigen Variablen.

48.3 Fakultät: eine Variation

Obige rekursive Implementierung der Fakultät ist ziemlicher Standard und überall zu finden. Mittels der bedingten Auswertung (der `?:`-Operator, siehe Kapitel 43) geht die Sache viel kürzer:

```
1 int fakultaet( int n )
2     {
3         return (n <= 1)? 1: n * fakultaet(n - 1);
4     }
```

Betrachtet dies als Lehrbeispiel für die Verwendung der verschiedenen Möglichkeiten in C.

48.4 Iterativ oder Rekursiv: eine Geschmacksfrage?

Eine gute Frage. Früher, zu Fortran-Zeiten, wurde üblicherweise von rekursiven Implementierungen abgeraten. Das hatte sicherlich auch seine guten Gründe. Beobachtet man in der heutigen Zeit unsere Studenten, stellt man folgendes fest: anfangs haben sie unwahrscheinlich Scheu davor, die Rekursion zu verwenden, da sie ihnen einfach undurchsichtig erscheint. Nach einer gewissen Eingewöhnungszeit haben sie Spaß daran und versuchen sie möglichst oft anzuwenden.

Dazu jetzt die Dozentensicht:

Regel 1: Wir erinnern uns, das Wichtigste ist und war, dass ein Programm richtig funktioniert. Die Frage ob nun rekursiv oder iterativ ist demnach eher zweitrangig.

Regel 2: Nimm das, was Dir lieber ist. Dann steigen auch die Chancen, dass Regel 1 erfüllt wird.

Regel 3: Wenn Regel 1 und 2 erfüllt sind, ist die Beantwortung der Frage eine Frage der benötigten Ressourcen. Bei jedem Funktionsaufruf, also auch bei jeder Rekursionsstufe, wird ein kompletter Stack Frame angelegt. Das kann sehr angenehm sein, kostet aber auch Arbeitsspeicher. Sollte die eigentliche Implementierung (also die Maschineninstruktionen) durch diese Zusatzkosten besonders klein und kompakt werden, lohnt es sicherlich. Wenn nicht, kostet es nur, tut aber vielleicht dem Ego gut (was hier nicht despektierlich gemeint ist!). Vor dem Hintergrund des Ressourcenverbrauchs ist es eher nicht sinnvoll, die Fakultät rekursiv zu implementieren. Aber sie ist dafür ein sehr gutes Lehrbeispiel. Sorry about that. Und für die Interessierten: In Kapitel 76 besprechen wir Beispiele, bei denen die Rekursion die Methode der Wahl ist.

Kapitel 49

Mehrdimensionale Arrays

Nachdem wir in Kapitel 33 eine erste Einführung in eindimensionale Arrays hatten, erfolgt in diesem Kapitel die Beschreibung mehrdimensionaler Arrays. Für die folgenden Ausführungen sind insbesondere folgende Kapitel sehr wichtig: 33, 45 und 46.

49.1 Vorbild: Matrizen in der Mathematik

Aus der Mathematik kennen wir den Begriff der (zweidimensionalen) Matrix. Eine einfache 2×3 Matrix sieht wie folgt aus:

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

Natürlich gibt es auch drei- und mehrdimensionale Matrizen, die man in der Regel auch Tensoren nennt. Diese mehrdimensionalen Matrizen (Tensoren) kann man 1:1 auch in der Programmiersprache C verwenden, wie wir in den folgenden Abschnitten sehen werden.

49.2 Verwendung

C-Syntax

```
double d[ 12 ][ 4 ];
int a[ 10 ][ 20 ];
a[ 0 ][ 3 ] = 4711;
a[ 9 ][ 19 ] = -1
```

Abstrakte Programmierung

```
Var.: Typ Array 0..11 × 0..3 of Double: d
Var.: Typ Array 0..9 × 0..19 of integer: a
setze a0,3 = 4711
setze a9,19 = -1
```

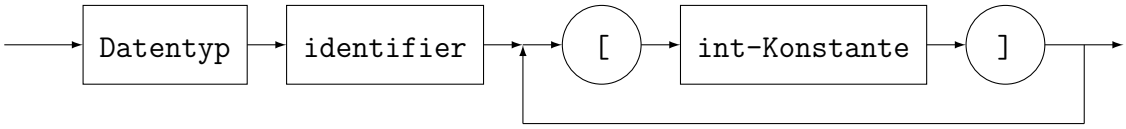
Hinweise: Bei der Verwendung von mehrdimensionalen Arrays sollte folgendes unbedingt beachtet werden:

1. Die Größe jeder einzelnen Dimension wird innerhalb eines gesonderten Paares eckiger Klammern `[]` angegeben.
2. Alle bisherigen Hinweise aus Kapitel 33 bezüglich erlaubter Indizes, Speicherbelegung und Zuweisungskompatibilität gelten hier sinngemäß.
3. Weder während der Übersetzung noch zur Laufzeit werden die Indizes auf Gültigkeit überprüft. Zugriffe mit ungültigen Indizes führen häufig zu sehr unerwünschten Effekten (da sich dort eventuell eine andere Variable befindet), manchmal auch zu einem Programmabsturz. Das Finden derartige Fehler ist in der Regel zeit- und nervenaufreibend.

49.3 Syntaxdiagramme

Das Syntaxdiagramm hatten wir bereits in Kapitel 33 wiederholen wir hier aber nochmals:

Definition n-dimensionales Array



49.4 Korrekte Beispiele

```
1 // Definitionen
2
3 int a[ 10 ][ 4 ];
4 double d[ 12 ][ 3 ];
```

```
1 // Zugriff
2
3 a[ 0 ][ 1 ] = 4711;
4 d[ 2 ][ 2 ] = 12.34;
```

49.5 Fehlerhafte Beispiele

```
1  // korrekte Definitionen
2
3  int a[ 10 ][ 4 ];        // ein 10 x 4 array
4  double d[ 12 ][ 3 ];     // ein 12 x 3 array
5
6  // fehlerhafte Definitionen
7
8  int m[ 3 ][ -1 ];        // zahl der elemente muss >= 0 sein
9  double n[ 12 ][ 0 ];     // korrekt, aber nicht sinnvoll,
10                          // da das array *kein* Element besitzt
11
12 // Zugriffe
13
```

```
14 a[ -1 ][ 0 ] = 1;      // falscher Index; erlaubt [0..9][0..3]
15 a[ 1 ] = 1;            // der zweite Index fehlt
16 d[ 1, 2 ] = 1;         // fehlendes Klammernpaar;
17                        // nicht [a,b] sondern [a][b]
18 d[ 1 ][ 1 ][ 0 ] = 1; // ein Index zu viel
19 d{ -1 }[ 0 ] = 1;      // falsche Klammern; nicht {} sondern []
```

49.6 Ausgabe eines Arrays

Auch mehrdimensionale Arrays müssen elementweise ausgegeben werden.

49.7 Interne Repräsentation

Die interne Repräsentation eindimensionaler Arrays im Arbeitsspeicher ist ziemlich einfach und klar: die Elemente werden der Reihe nach im Arbeitsspeicher angeordnet. Aber wie ist das nun bei mehrdimensionalen Arrays? Es gibt zwei Möglichkeiten, zeilenweise und spaltenweise. C plaziert die Elemente zeilenweise[1]. Der folgende Quelltext zeigt die Definition eines `2x2` Arrays sowie dessen Anordnung im Arbeitsspeicher (niedrige Adressen links, hohe Adressen rechts):

```
1 // Deklaration           // Arbeitsspeicher
2    int a[2][2];          // a[0][0] a[0][1] a[1][0] a[1][1]
```

Wie man gut sehen kann, ändert sich der Spaltenindex schneller als der Zeilenindex. Mit anderen Worten, es wird immer erst eine ganze Zeile plaziert, bevor die nächste Zeile genommen wird. Die Erweiterung auf mehr als zwei Dimensionen sollte offensichtlich sein. Wer hier etwas zweifelt, schreibe sich einfach ein kleines Testprogramm, das die Adressen der Elemente eines mehrdimensionalen Arrays ausgibt.

49.8 Deklaration einschließlich Initialisierung

Wie eindimensionale Arrays (siehe auch Kapitel 33.8) können auch mehrdimensionale Arrays bei ihrer Definition gleichzeitig initialisiert werden. Das folgende Programmstück zeigt ein kleines Beispiel.

```
1 // Definition
2    int a[2][3] = { 1, 2, 3, 4, 5, 6 };
3
4 // Resultat
5 // a[0][0] = 1, a[0][1] = 2, a[0][2] = 3,
6 // a[1][0] = 4, a[1][1] = 5, a[1][2] = 6
```

[1] Nicht alle Programmiersprachen machen dies so. Fortran beispielsweise organisiert seine Arrays spaltenweise.

Wie am Resultat zu sehen ist, erfolgt die Initialisierung gemäß der Anordnung der einzelnen Elemente im Arbeitsspeicher.

Da diese erste Form bei größeren Arrays recht unübersichtlich und wenig änderungsfreundlich ist, kann man die Initialisierungen auch wie folgt gruppieren:

```
1 // Definition
2    int a[2][3] = { {1, 2}, {4, 5, 6}};
3
4 // Resultat
5 // a[0][0] = 1, a[0][1] = 2, a[0][2] = undefiniert,
6 // a[1][0] = 4, a[1][1] = 5, a[1][2] = 6
```

Dieses Beispiel zeigt auch folgendes: Da in der ersten Zeile der dritte Wert fehlt, ist das Element `a[ 0 ][ 2 ]` undefiniert.

49.9 Größenfestlegung durch Initialisierung

Hier gilt analog das, was wir in Kapitel 33.9 über eindimensionale Arrays gesagt haben. Ferner können wir die Strukturierungshilfe aus dem vorherigen Abschnitt verwenden.

49.10 Größen einzelner Teil-Arrays

Die folgende Tabelle zeigt, wie der Compiler sehr intuitiv die Größen einzelner Teil-Arrays berechnet. Dabei sind alle Größenangaben als Vielfache der Größe eines `int` ausgedrückt; oder einfach: 1 ≙ `sizeof(int)`.

Definition: `int a[2][3][4];`

Ausdruck	Größe	Kommentar
`sizeof(a)`	24	Das gesamte Array
`sizeof(a[ 1 ])`	12	Die zweite Zeile
`sizeof(a[ 1 ][ 0 ])`	4	Die erste Spalte der zweiten Zeile
`sizeof(a[ 1 ][ 0 ][ 2 ])`	1	Ein einzelnes Element

49.11 Mehrdimensionale Arrays als Parameter

Zum Schluss klären wir noch die Frage, wie man mehrdimensionale Felder an eine Funktion übergibt. Bei eindimensionalen Feldern ist der Array-Name gleich der Anfangsadresse des Arrays und alle Elemente sind der Reihe nach im Arbeitsspeicher abgelegt.

Wie wir oben gelernt haben, werden auch mehrdimensionale Arrays im Arbeitsspeicher linear abgelegt, und zwar Zeile für Zeile. Um den Ort eines Elementes wiederzufinden,

muss die Funktion (mit Hilfe des Compilers) wissen, wie die Struktur einer Zeile ist bzw. sie muss wissen, wie lang eine Zeile ist. Aus diesem Grund *muss* man mit Ausnahme der ersten die Größen aller Dimensionen im Funktionskopf spezifizieren:

```
 1 int f( int a[][3][2], int size )
 2     {
 3         // jetzt findet f das Element a[1][2][0] ohne die
 4         // Spezifikation [3][2] waere dies nicht moeglich
 5     }
 6
 7 int main( int argc, char **argv )
 8     {
 9        int x[ 2 ][ 3 ][ 2 ];
10        f( x, 2 );
11        f( x, sizeof( x )/sizeof(x[ 0 ]) );
12     }
```

Das Programm sollte recht selbsterklärend sein. Die Funktion `f()` macht natürlich nichts, sie dient nur als Beispiel. Durch die Größenangaben für die zweite und dritte Dimension `int a[][ 3 ][ 2 ]` kennt die Funktion `f()` das Array-Layout und kann auf jedes Element `a[ i ][ j ][ k ]` mittels `RAM[ 6*i + 2*j + k ]` im Speicher zugreifen.

Gegebenenfalls lohnt sich noch ein vergleichender Blick auf die beiden Zeilen 10 und 11: Beide sind funktionell identisch, Zeile 11 hingegen wesentlich änderungsfreundlicher. Warum beim zweiten Ausdruck auch der Wert `2` heraus kommt, sollte inzwischen eigentlich klar sein, wenn nicht hilft die Tabelle des vorherigen Abschnitts weiter.

Alternativ hätte man die Funktion `f()` auch wie folgt definieren können: `int f( int *p, int sx, int sy, int sz )`. Nur hätte man dann die richtigen Indizes bei Speicherzugriffen `p[ x*sy*sz + y*sz + z ]` selbst ausrechnen müssen, was nicht unbedingt änderungsfreundlich ist.

Kapitel 50

Zeichenketten bzw. Datentyp `string`

Nach den ganzen vielen Arrays, Zeigern und Adressberechnungen kommen wir endlich zu den Zeichenketten, die im Englischen auch als *Strings* bezeichnet werden. Das wir dieses Thema genau jetzt aufgreifen ist kein Zufall. Zeichenketten werden in C *üblicherweise* als `char`-Arrays abgelegt, die am Ende ein zusätzliches Null-Byte `'\0'` bzw. einfach `0` haben. Im Gegensatz zu anderen Programmiersprachen gibt es in C keinen gesonderten Datentyp `string`. Und klar, wenn Zeichenketten als `char`-Array abgelegt werden, ist ihr Datenyp `char *`.

Zeichenketten sind für die sinnvolle (interaktive) Ein-/Ausgabe ein unerlässliches Hilfsmittel. Daher versuchten die Sprachentwickler den Umgang mit ihnen möglichst einfach zu gestalten. Der Compiler erkennt konstante Zeichenketten daran, dass sie in Anführungszeichen " eingeschlossen sind. Diese Anführungszeichen werden vom Compiler entfernt. Anschließend wird noch das Null-Byte angehängt, damit alle Standardfunktionen wissen, wo die Zeichenkette zu Ende ist. Mit anderen Worten, bei Zeichenketten wird nicht wie sonst notwendig, die Größe des Arrays übergeben, sondern das Ende wird durch einen speziellen Marker erkannt, der sowieso keinem vernünftigen druckbaren Zeichen entspricht. Wichtig ist, dass im Regelfall konstante Zeichenketten während der Programmlaufzeit *nicht verändert* werden können und dürfen; entsprechende Versuche führen im Regelfall zum Programmabbruch.

Für das weitere Verständnis sind neben den Kapiteln 29 und 30 insbesondere die Kapitel 40, 45 und 46 Voraussetzung.

50.1 Verwendung

C-Syntax

```
"hello, world\n"
printf( "a=%d", a );
```

Abstrakte Programmierung

```
Text: hello, world\n
Drucke: Text: "a=%d"Variable a
```

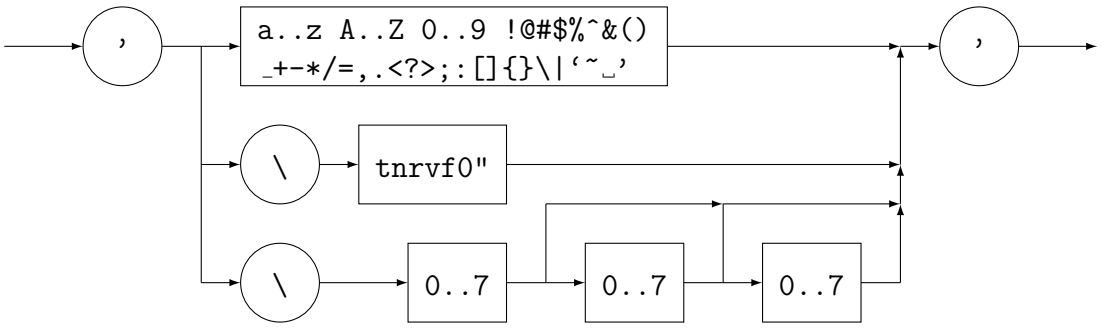
Hinweise: Bei der Verwendung von *Zeichenketten* sollten folgende Dinge unbedingt beachtet werden:

1. Am Anfang und am Ende einer Zeichenkette muss jeweils ein Anführungszeichen stehen. Diese beiden Anführungszeichen werden für die interne Repräsentation vom Compiler entfernt! Dafür wird am Ende *immer* ein Null-Byte angehängt.
2. Soll innerhalb einer Zeichenkette ein Anführungszeichen erscheinen, so ist diesem ein Backslash voranzustellen. Beispiel: `"very \"smart\" move"` wird intern zu: `very "smart" move`.
3. Innerhalb der Zeichenkette können beliebig viele ASCII-Zeichen stehen. Diese können kodiert werden, wie wir in den Kapiteln 29 und 30 beschrieben haben. Es sind also „normale“ Zeichen (`A`, `M`), Escape Sequenzen (`\n`, `\t`) sowie die oktale Ersatzdarstellung (`\101`) erlaubt.
4. Zeichenketten dürfen nicht über das Zeilenende hinaus gehen; Ausnahme: vor dem Zeilenende steht ein Backslash.
5. *Konstante* Zeichenketten kann man auch trennen/verknüpfen. Die beiden folgenden Programmzeilen sind identisch, da der Compiler die Teil-Strings verbindet, solange sich *nur* Leerzeichen, Tabulatoren und Zeilenwechsel zwischen ihnen befinden.

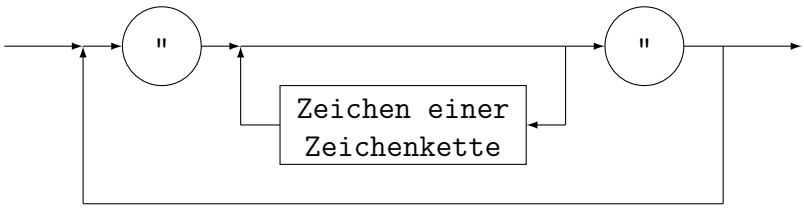
```
1 printf( "Hello" " world," " I am" " fine" "\n" );
2 printf( "Hello world, I am fine\n" );
```

50.2 Syntaxdiagramme

Zeichen einer Zeichenkette



konstante Zeichenkette



50.3 Korrekte Beispiele

```
1 char *p;                  // ein Zeiger auf eine Zeichenkette
2 p = "Hello, world";       // der Klassiker ;-)
3 p = "Hello, world\n";     // dito, nur mit einem Zeilenumbruch
4 p = "\ti=%10d\ts='%s'"; // tabellarische Formatierung
5 p = "abc\"def";           // Zeichenkette mit Anfuehrungszeichen
6 p = "012345";             // sechs Ziffern
7 p = "\06012345";          // dito aber 0 als Escape-Sequenz \060
8 p = "ABCD";               // vier Buchstaben
9 p = "\101\102\103\104"; // dito, aber Ersatzdarstellung
```

50.4 Fehlerhafte Beispiele

```
1 char *p;                  // ein Zeiger auf eine Zeichenkette
2 p = "Hello, world;        // zweites Anfuehrungszeichen fehlt
3 p = "abc"def";            // Backslash in der Mitte vergessen
4 p = 'hi there';           // Apostroph statt Anfuehrungszeichen
```

50.5 Interne Repräsentation

Wie aus obigen Erläuterungen klar geworden sein sollte, besteht eine (konstante) Zeichenkette aus einer Aneinanderreihung einzelner Zeichen, die ihrerseits vom Typ `char` sind. Entsprechend werden konstante Zeichenketten vom Compiler intern als Arrays behandelt, die *keinen* expliziten Namen haben; man bezeichnet sie daher auch als anonyme Arrays. Folgende Abbildung zeigt beispielhaft die interne Repräsentation der Zeichenkette `"Hi there"`:

String:	"Hi there"								
Array :	'H'	'i'	' '	't'	'h'	'e'	'r'	'e'	'\0'
Index :	0	1	2	3	4	5	6	7	8

Beim Betrachten der internen Repräsentation fällt folgendes auf: Die Zeichenkette besteht aus acht Zeichen, die interne Repräsentation aber aus neun Zeichen. Der Compiler hat *von sich aus* das zusätzliches Zeichen `'\0'` an die Zeichenkette angehängt, das den Array-Index acht erhält. Der Grund hierfür ist sehr einfach: an jeder beliebigen Stelle im Programm, beispielsweise innerhalb der Funktion `printf()`, ist es in einfacher Weise möglich, das Ende der Zeichenkette zu finden, da per Definition das Null-Byte die Zeichenkette beendet; problematisch wird es, wenn das Null-Byte aufgrund eines Programmierfehlers fehlt, da es dann zu unsinnigen Berechnungen oder Programmabstürzen kommen kann.

Es ist sogar so, dass alle Bibliotheksfunktionen, die mit Zeichenketten hantieren, dieses Null-Byte am Ende der Zeichenkette erwarten und davon ausgehen, dass dieses Null-Byte lediglich als Endekennung fungiert und nicht zur eigentlichen Zeichenkette gehört. Letzt-

lich ist es sogar so, dass Funktionen wie `puts()` eine Zeichenkette wie folgt zeichenweise ausgeben:

```
1  int puts( char *p )
2      {
3         while( *p != '\0' )
4            putc( *p++, stdout );
5      }
```

Größe und Länge von Zeichenketten: Die Begriffe *Größe* und *Länge* einer Zeichenkette dürfen nicht miteinander verwechselt werden, da diese in der Programmiersprache C zwei gänzlich unterschiedliche Dinge sind. Als Länge einer Zeichenkette bezeichnet man die Zahl der „sinnvollen“ Nutzzeichen. Bei der Zeichenkette `"Montag"` sind dies beispielsweise sechs Zeichen, wie auch von der Bibliotheksfunktion `strlen()` bestätigt wird. Hingegen bezeichnet die Größe einer Zeichenkette die Zahl der Zeichen innerhalb des Arbeitsspeichers. Diese ist, wie oben ausgeführt, um eins größer als die Länge der Zeichenkette, da ja der Compiler von sich aus noch besagtes Null-Byte anhängt. In der Tat liefert der Funktionsaufruf `sizeof( "Montag")` den Wert sieben zurück.

Ausgeben von Null-Bytes `'\0'`: Bei Betrachtung des Null-Bytes ensteht die Frage, ob man Null-Bytes *sinnvoll* in eine Zeichenkette einfügen kann. Die klare Antwort ist: im allgemeinen *nein* (siehe aber auch Abschnitt 50.6). Dies ist auch keine Einschränkung, da sich an dieser Stelle in der ASCII-Tabelle kein sinnvolles Zeichen befindet. Wer dies nicht glaubt, kann einmal folgendes Programm ausprobieren:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv )
4      {
5         printf( "-->'%c'<--\n", (char) '\0' );
6      }
```

In jedem Fall wird ein 'Null-Byte' ausgegeben. Je nach Terminaleinstellungen sieht man entweder nichts oder ein Sonderzeichen. Lenkt man die Ausgabe in eine Datei um, kann man – je nach Terminaleinstellung – etwas im Editor sehen.

50.6 Zeichenketten mit Null-Bytes

Entgegen obigen Ausführungen kann es für einen Programmierer doch sinnvoll sein, Null-Bytes in seine Zeichenketten einzufügen. Beispielsweise könnten verschiedene Namen wie bei uns üblich aus Vor- und Nachname bestehen. Mit der richtigen Ausgabefunktion kann man diese Zeichenketten sinnvoll auswerten. Hierzu folgendes kleines Beispiel:

```
1  #include <stdio.h>
2  #include <string.h>              // for: int strlen( char *p )
3  #define enrico  "Enrico\0Heinrich"          // just an example
```

```
4
5  void print( char *p )
6       {
7          printf( "Vorname= %-9s Nachname= %s\n",
8                                  p, p + strlen( p ) + 1 );
9       }
10 int main( int argc, char **argv )
11     {
12        print( enrico );
13        print( "Matthias\0Hinkfoth" ); print( "Ralf\0Joost" );
14        print( "Ralf\0Salomon" );    print( "Ralf\0Warmuth" );
15     }
```

Dieses Programm erzeugt folgende Ausgabe:

```
1  Vorname= Enrico    Nachname= Heinrich
2  Vorname= Matthias  Nachname= Hinkfoth
3  Vorname= Ralf      Nachname= Joost
4  Vorname= Ralf      Nachname= Salomon
5  Vorname= Ralf      Nachname= Warmuth
```

50.7 Besonderheit: sehr lange Zeichenketten

Manchmal können Zeichenketten sehr lang werden. Beispielsweise weil man viel sagen möchte oder weil man mehrere Zeilen mittels eines `printf()` ausgeben möchte. Hierfür gibt es eine schöne Möglichkeit, die bereits in obigem Syntaxdiagramm angedeutet ist: Tauchen im Quelltext nacheinander mehrere Zeichenketten auf, zwischen denen sich *nur* Leerzeichen, Tabulatoren und Zeilenumbrüche befinden, „klebt“ der Compiler diese Zeichenketten selbstständig zu einer Zeichkette zusammen. Beispiel? Here we go:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4      {
5         char *str = "abc" "def"
6         "ghi";
7         printf( "'%s'\n", str );
8      }
```

Dieses Programm produziert tatsächlich die Ausgabe `'abcdefghi'`.

50.8 Besonderheit: Ausgabe von Zeichenketten

Normalerweise geben wir alle Texte und Variablenwerte mittels `printf()` aus. Wenn in der ersten Zeichenkette %-Sequenzen auftauchen, werden diese durch die aktuellen Werte der nachfolgenden Parameter ersetzt. Beispiel: `i = 2; printf( "i= %d\n", i )` erzeugt die Ausgabe `i= 2`. Soweit, so gut.

Nehmen wir nun an, wir hätten folgendes Programmstück:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4     {
5         char *str = "Montag";
6         printf( str );
7     }
```

Dann käme wie zu erwarten auch wirklich `Montag` heraus. Aber was würde passieren, wenn wir `str = "Hallo %d hihi"` hätten? Auf jeden Fall nichts Gutes, denn wir haben bei unserem `printf( str )` kein weiteres Argument. Der Funktion `printf()` ist es nämlich egal, ob der erste Parameter direkt in den Funktionsaufruf hinein geschrieben wird, oder ob dieser von einem Zeiger kommt. Und wenn nun die Funktion `printf()` die Zeichenkette `"Hallo %d hihi"` ausgeben soll, bekommen wir ein Problem ab dem siebten Zeichen, denn hier erwartet sie ein weiteres Argument (auf dem Stack), was gar nicht da ist. Es kommt zumindest zu merkwürdigen Ausgaben oder zum Programmabsturz. Aus diesem Grund sollten Zeichenketten lieber mittels `printf( "%s", str )` ausgegeben werden, denn ab dem zweiten Argumenten führt `printf()` keinerlei Ersetzung mehr durch. Mit anderen Worten: Die Anweisung `printf( "%s", "%d")` gibt tatsächlich `%d` aus.

50.9 Speichersegmente

Wo was im Arbeitsspeicher angelegt wird, sollte mittlerweile klar sein.

Wie der Name schon sagt, sind konstante Zeichenketten Konstanten. Entsprechend werden sie alle im Konstanten-Segment abgelegt, wie wir bereits in Kapitel 40, Seite 151 besprochen haben. Wenn wir aber einen Zeiger auf eine konstante Zeichenkette haben, wird diese Variable am üblichen Ort abgelegt. Verwirrt? Dann betrachten wir mal folgendes Programmstück:

```
1 int main( int argc, char **argv )
2     {
3         char *str = "Montag";
4     }
```

Die Zeichenkette `"Montag"` wird im Konstanten-Segment abgelegt, wohingegen der Zeiger `str` auf dem Stack zu finden ist.

Zugriffsbeschränkungen: Um es nochmals zu wiederholen: konstante Zeichenketten sind *Konstanten*. Entsprechend dürfen bzw. können sie nicht verändert werden. Betrachten wir hierzu folgenden Programmausschnitt:

```
1  int main( int argc, char **argv)
2      {
3          char *p = "Montag";
4          p[ 1 ] = 'x';
5      }
```

In der Regel führt Zeile 4 zu einem Programmabsturz[1].

Soll eine konstante Zeichenkette tatsächlich mal verändert werden, muss zunächst eine lokale Kopie erzeugt werden. Dieses geht beispielsweise mittels eines „ordentlichen“ Arrays oder unter Zuhilfenahme eines Aufrufs der Funktion `malloc()`, die wir aber erst in Kapitel 69 besprechen.

50.10 Zeichenketten zur Initialisierung von Arrays

Konstante Zeichenketten können wie andere Konstanten (also Zahlen oder Zeichen) zur Initialisierung von Arrays verwendet werden. Für das tiefere Verständnis muss man sich immer vergegenwärtigen, wie konstante Zeichenketten im Arbeitsspeicher abgelegt werden. Hierzu zwei kleine Beispiele:

```
1  int main( int argc, char **argv )
2      {
3          char a[] = "abc";
4          char b[] = { 'a', 'b', 'c' };
5      }
```

In diesem Beispiel werden zwei Arrays deklariert, `a[]` und `b[]`. Im ersten Fall besteht der „Initialisierungsstring“ aus genau vier Zeichen, nämlich `'a'`, `'b'`, `'c'` und `'\0'`. Entsprechend hat das Array `a[]` vier Elemente, in denen die vier Zeichen stehen. Dieses Array kann also wie eine ganz „normale“ Zeichenkette verwendet werden.

Im Gegensatz dazu stehen für die Initialisierung des Arrays `b[]` nur drei Elemente zur Verfügung, sodass auch das Feld nur aus drei Elementen besteht, in denen der Reihe nach die Zeichen `'a'`, `'b'` und `'c'` stehen. Da das Null-Byte nicht vorhanden ist, kann das Array `b[]` auch *nicht* wie eine „normale“ Zeichenkette verwendet werden.

[1] Zumindest gilt dies für Programme, die mittels eines Betriebssystems wie Linux oder Windows gestartet werden, da ein ungültiger Speicherzugriff 'Schreiben in ein read-only Segment' vorliegt.

50.11 Programmbeispiele

Dieser Abschnitt diskutiert einige Beispiele, um den Umgang mit Zeichenketten etwas zu festigen. Diese Beispiele findet man in der einen oder anderen Form auch in der einschlägigen Literatur [4].

Länge einer Zeichenkette: Die Länge einer Zeichenkette kann wie folgt ermittelt werden:

```
1  int strlen( char str[] )
2      {
3         int len;
4         for( len = 0; str[ len ]; len++ )
5            ;
6         return len;
7      }
```

In dieser Variante wird die Variable `len` für jedes Zeichen, das ungleich dem Null-Byte ist, um eins erhöht. Dies bedeutet, dass bei der Bestimmung der Länge der Zeichenkette das Null-Byte nicht berücksichtigt wird; dies ist auch bei den Funktionen der Standardbibliothek so. Unter Verwendung von Zeigern sieht die Funktion `strlen()` wie folgt aus:

```
1  int strlen( char *p )
2      {
3         int len;
4         for( len = 0; *p; p++ )
5            len++;
6         return len;
7      }
```

Eine alternative Implementierung könnte wie folgt aussehen:

```
1  int strlen( char *p )
2      {
3         char *q = p;
4         while( *p )
5            p++;
6         return p-q;
7      }
```

Bei dieser Implementierung wird der Anfang der Zeichenkette im Hilfszeiger `q` abgelegt. Dann wird der Zeiger `p` so lange weitergeschaltet (Zeile 5), bis das Null-Byte gefunden wurde. Am Ende (Zeile 6) wird die Länge durch die Adressdifferenz der beiden Zeiger `p` und `q` ermittelt.

Je nach Prozessor und Compiler könnte aus Effizienzgründen auch folgende Implementierung sinnvoll sein:

```
1 int strlen( char *p )
2     {
3         char *q = p;
4         while( *p++ )
5             ;
6         return --p - q;
7     }
```

denn einige Prozessoren können das Abfragen des Inhaltes (`*p`) und das Weiterschalten (`p++`) in einer Maschineninstruktion abarbeiten. In Zeile 6 muss hier der Zeiger `p` wieder um eins zurückgesetzt werden, da sonst das Null-Byte mitgezählt würde.

50.12 Akademische Hintergrunddiskussion

Im Laufe ihrer Ausbildung stellen sich früher oder später viele C-Programmier die folgenden beiden Fragen:

1. Warum gibt es in der Programmiersprache C eigentlich keinen Datentyp `string`?
2. Warum gibt es dieses „komische“ Null-Byte `'\0'`?

Zu Frage 1: Die Antwort auf die erste Frage ist zweigeteilt. Die erste Teilantwort könnte lauten: In jener Zeit, als die Programmiersprache C sowie die ersten Unix-Kernel entstanden, war der Bedarf an der direkten Verarbeitung von Zeichenketten nicht so bedeutend wie heute. Insofern war auch der Druck, dies mit Mitteln der Programmiersprache zu realisieren, nicht so groß.

Die zweite Teilantwort ist schwieriger und erschließt sich den meisten Lesern nicht so ohne weiteres, da hierfür ein tieferes Verständnis von Betriebssystemen und Rechnerorganisation vonnöten ist. Beispielsweise muss beim Zusammenfügen zweier Zeichenketten genug freier Speicherplatz zur Verfügung stehen. Dies müsste der Betriebssystemkern alleine bewerkstelligen, was unter Umständen nicht so einfach geht. Insofern, als einfaches Beispiel, wäre die Integration von Operationen auf Zeichenketten zumindest nicht unproblematisch.

Zu Frage 2: Normalerweise gilt, dass man die Größe eines Arrays in dem Block bestimmen muss, in dem das Array deklariert (also erzeugt) wurde. Diese Größe müsste man aber immer durch alle Funktionen mitführen, damit man die Größe weiß. Diese Vorgehensweise wäre aber bei konstanten Zeichenketten zumindest sehr mühsam. Hinzu kommt, dass die Arrays, in denen die Zeichenketten abgelegt werden, keinen Namen haben; ensprechend müsste man die (konstanten) Zeichenketten ein zweites Mal verwenden, nämlich bei einem Aufruf von `sizeof()`. Durch die Verwendung des Null-Bytes kann man aber immer, egal an welcher Stelle, bestimmen, wie lang eine Zeichenkette ist. Vor diesem Hintergrund ist die Verwendung des Null-Bytes recht angenehm. Hinzu kommt, dass die ASCII Tabelle an der Stelle null kein sinnvolles Zeichen hat. Insofern handelt es sich auch aus dieser Perspektive nicht um eine Einschränkung.

Alternative Designentscheidungen: Eine alternative Designentscheidung hätte darin bestanden, die Länge der Zeichenketten direkt in die Zeichenkette zu integrieren. Dies ist beispielsweise bei Turbopascal so gemacht worden. Allerdings wäre dann die Frage gewesen, wo man genau die Größe abspeichert. Eine Lösung bestünde beispielsweise darin, die Länge in einem Byte vor der eigentlichen Zeichenkette zu speichern. Auch wenn dies so funktionieren würde, hätte dieser Ansatz dennoch einige Problemstellen: die Länge der Zeichenkette könnte fehlerhaft werden, wenn man versehentlich dieses Byte verändert, die Länge von Zeichenketten wäre auf 255 beschränkt, was zumindest heutzutage einer echten Einschränkung entspräche, um mal zwei Beispiele zu nennen.

Letztlich könnte man Zeichenketten auch in Form eines `structs` (siehe Kapitel 53) definieren, in dem eine Komponente die Länge, eine zweite die eigentliche Zeichenkette repräsentieren würde. Aber auch dies ist zumindest für viele Anwendungsfälle recht unhandlich; allerdings findet man diesen Ansatz in Programmiersprachen wie C++ Java und der Qt-Bibliothek wieder.

Zwang oder Empfehlung? Eine weitere Frage, die nach der Diskussion auftauchen könnte, ist, ob man Zeichenketten wie oben beschrieben ablegen und verarbeiten muss. Die Antwort lautet natürlich *nein*! Nur kann man dann keine der Standardfunktionen wie `printf()`, `strcmp()` etc. mehr verwenden; im Gegenteil, man müßte alles selbst von neuem programmieren. Aber wer will das schon ...

Kapitel 51

Kommandozeile: `argc` und `argv`

Wo wir doch gerade dabei sind, könnten wir endlich mal über die beiden Parameter `argc` und `argv` des Hauptprogramms reden. Der eine oder andere wird sich sicherlich schon seit einiger Zeit gefragt haben, warum wir die `main()`-Funktion immer so umständlich implementieren, vor allem weil viele Kumpels, Bücher und Leerunterlagen doch einfach nur `main(void)` schreiben. Ist doch viel kürzer, warum der Aufwand? Genau das klären wir hier mal eben ;-)

51.1 Hintergrund

Auch bei diesem Thema gilt wieder, dass die Programmiersprache C sehr eng mit der Entwicklung des Unix Betriebssystems verbunden ist. Zur Erinnerung: Die Programmiersprache C wurde entwickelt, um Unix in einer Hochsprache zu implementieren. Und mit der Entwicklung von Unix wurden viele neue Konzepte eingeführt, wozu auch die beiden Parameter `argc` und `argv` gehören. Sie stellen die Verbindung zwischen Programm und Kommandozeile her, da man mit ihnen ein Kommando mit Argumenten versorgen kann.

Ja, Kommandos und Parameter stammen aus einer Zeit, in der man froh war, wenn man einer von 20 war, der mittels eines VT100 Terminals direkt an einem kleinen Rechner arbeiten konnten. Und ja, damals haben alle 20 parallel an einem Rechner gearbeitet. Und es hat funktioniert! Und durch den `argc/argv`-Mechanismus weiß beispielsweise der Compiler, welche Datei er übersetzen und wo er das Ergebnis hinschreiben soll.

Und nein, weder ist es in der heutigen Zeit anders, noch ist es überflüssig. Wenn Ihr beispielsweise ein Word-Dokument `Klausur.doc` auf Eurem Desktop liegen habt und es mittels eines raffinierten Doppel-Klicks öffnet, dann ruft das Betriebssystem (die Shell) Euer Word-Programm wie folgt auf: `ooffice ~/Desktop/Klausur.doc` (Linux). Durch diesen Mechanismus weiß das Textverarbeitungsprogramm, welche Datei Ihr bearbeiten wollt; der `argc/argv`-Mechanismus ist also noch präsent, nur merkt Ihr es nicht.

51.2 Funktionsweise im Überblick

Dieser Abschnitt erklärt erst einmal die grobe Funktionsweise auf einer phänomenologischen Ebene, d.h. wir sprechen erst einmal darüber, was so passiert, ohne uns um die einzelnen Implementierungsdetails zu kümmern. *„Wie jetzt? Warum erst mal wieder so ungefähr?“* Ganz einfach, die Sache ist beim ersten Mal wieder recht kompliziert, da so viele Dinge und Mechanismen involviert sind. Versuchen wir es mal wie folgt: Der Aufruf eines Programms mittels der Kommandozeile erfolgt grob in den folgenden fünf Schritten:

1. Zunächst einmal tippt der Benutzer ein Kommando nebst seiner Argumente ein. Das könnte beispielsweise unser C-Compiler sein:

   ```
   gcc -o mega mega.c
   ```

   oder die Eingabe der Zeile:

   ```
   echo hi 123 456 789
   ```

   Hinweis: `echo` gibt lediglich seine Argumente aus und ist damit eine geniale Testhilfe insbesondere bei der Erstellung von Shell-Scripten (Linux) und Batch-Dateien.

2. Die Shell liest erst einmal die gesamte Zeile ein und zerlegt sie in ihre einzelnen Bestandteile. Diese Bestandteile sind die einzelnen *Wörter*, die üblicherweise eine Folge von Zeichen sind. Normalerweise darf ein Wort weder ein Leerzeichen, noch einen Tabulator oder Zeilenumbruch (die Entertaste) enthalten. Doch mehr dazu in Abschnitt 51.6. In unseren beiden obigen Beispielen haben wir folgende Bestandteile:

Beispiel	Wort 1	Wort 2	Wort 3	Wort 4	Wort 5	Parameter (`argc`)
1	`gcc`	`-o`	`mega`	`mega.c`	—	4
2	`echo`	`hi`	`123`	`456`	`789`	5

   Die Gesamtzahl der Wörter (letzte Spalte) wird uns später als `argc` wieder begegnen.

3. Im nächsten Schritt werden alle Wörter der Eingabezeile in ein Array gepackt. Da es sich bei den Elemeten um Zeichenketten handelt, sind diese vom Typ `char *`. Entsprechend ist das Array als `char *argv[]` definiert (siehe auch weiter unten).

4. Per Definition ist das erste Wort immer der Name des auszuführenden Programms. Die Shell (die Kommandoeingabe bzw. der Kommandointerpreter) sucht nun eine Datei gleichen Namens in dem sie der Reihe nach verschiedene Dateiverzeichnisse durchsucht, die üblicherweise in einer Umgebungsvariablen wie `$PATH` definiert sind.

5. Sollte die Shell eine entsprechende Datei gefunden haben, wird ihr vollständige Pfadname nebst dem Array mit allen Wörtern der Kommandozeile mittels eines System Calls (beispielsweise `execve()` in Linux) an das Betriebssystem übergeben. Dieser System Call sorgt nun dafür, dass das ausführbare Programm in den Arbeitsspeicher geladen und über `argc/argv` mit den Argumenten der Eingabezeile versorgt wird.

Sofern die `main()`-Funktion ordnungsgemäß als `int main( int argc, char **argv )` definiert wurde, ergeben sich bei unseren beiden obigen Beispielen folgende Zuordnungen:

argc	argv[0]	argv[1]	argv[2]	argv[3]	argv[4]
4	gcc	-o	mega	mega.c	
5	echo	hi	123	456	789

Auf die einzelnen Bestandteile können wir nun (verblüffend) einfach über das Array und den entsprechenden Index zugreifen. Dies schauen wir uns im nächsten Abschnitt anhand eines kleinen Programmierbeispiels noch einmal genauer an.

51.3 Kleines Programmbeispiel für `argc/argv`

Zur Erinnerung: Die einzelnen Argumente des Programmaufrufs werden in einem Array abgelegt, dessen Elemente vom Typ `char *` sind. Die Anfangsadresse dieses Arrays wird an die `main()`-Funktion übergeben. Da es sich dabei um einen formalen Parameter einer Funktion handelt, ist dieser entweder als `char **argv` bzw. `char *argv[]` definiert, was äquivalent ist. Und wie bei Arrays üblich (bzw. notwendig), muss auch die Größe dieses Arrays übergeben werden, wofür der üblicherweise als `int argc` bezeichnete Parameter zuständig ist. Daher lautet der korrekte Funktionskopf der `main()`-Funktion *immer*:

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

Auf die einzelnen Bestandteile des Arrays können wir völlig ungeniert zugreifen. Beispielsweise `argv[0]`, `argv[1]`, `argv[2]`, `argv[3]` usw. Da der Parameter `argc` die Größe des Arrays angibt, sind als Indizes `argv[ 0 ]` bis `argv[argc - 1]` erlaubt. Da übrigens jedes Kommando einen Namen hat, hat `argc` mindestens immer den Wert 1, also `argc` $\geq$ `1`. Natürlich können wir auf die einzelnen Argumente auch über die übliche Zeigernotation zugreifen: `*argv`, `*(argv+1)`, `*(argv+2)`, `*(argv+3)` usw.; zur Erinnerung: beide Notationen sind absolut äquivalent.

Da jedes einzelne Argument eine Zeichenkette ist, können wir auf die einzelnen Zeichen in der üblichen Array- bzw. Zeigernotation zugreifen. Beispielsweise erhalten wir das vierte Zeichen des ersten Wortes durch `argv[0][3]`, `(*argv)[3]` oder auch `*((*argv)+3)`. In obigem `echo`-Beispiel wäre dies immer das Zeichen `'o'`.

Neben der Versorgung eines Programms mit (den notwendigen) Parametern eignet sich dieser `argc/argv`-Mechanismus besonders gut zum Testen von Programmen. In der Kommandozeile können wir entsprechende Parameter übergeben, mittels derer eine neue Funktion oder ein neuer Algorithmus getestet werden kann. Im Folgenden besprechen wir ein kleines Beispiel, in dem wir eine Funktion `my_strlen()` zur Berechnung der Länge einer Zeichenkette entwickelt haben:

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

```
 2
 3 int my_strlen( char *p )
 4     {
 5        int len = 0;
 6        while( p[ len ] )
 7           len++;
 8        return len;
 9     }
10
11 int main( int argc, char **argv )
12     {
13        int i;
14        for( i = 0; i < argc; i++ )
15           printf( "my_strlen( %s ) = %d\n",
16                   argv[ i ], my_strlen( argv[ i ] ) );
17        return 0;
18     }
```

Ein Programmaufruf `./my-strlen windsurfing on maui` liefert:

```
1 my_strlen( ./my_strlen ) = 11
2 my_strlen( windsurfing ) = 11
3 my_strlen( on ) = 2
4 my_strlen( maui ) = 4
```

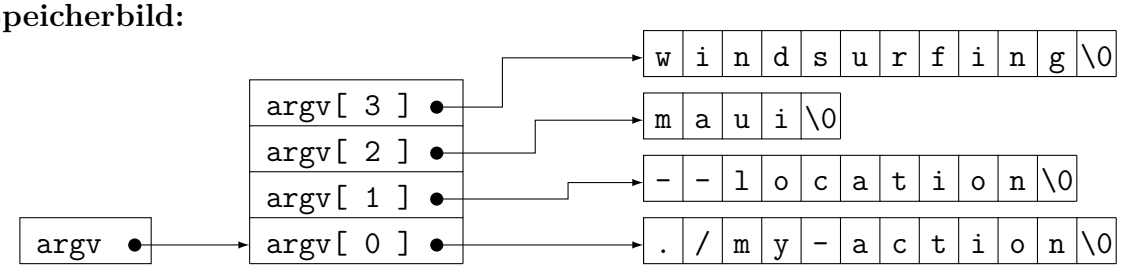
„Mensch, wenn man sich mal ein paar grundlegende Dinge klar gemacht hat, ist die Sache gar nicht mehr so schwer ;-)*“* Genau! Aber ein paar Details müssen wir uns noch ansehen.

51.4 Interne Repräsentation

Für das bessere Verständnis (einiger Details) sollten wir uns noch die interne Repräsentation anschauen. *„Puhhh, ist doch überflüssig, mir ist doch alles klar!“* Schauen wir mal, schaden kann es ja nichts. Schauen wir uns doch einfach mal folgendes Beispiel an:

Eingabe: `./my-action --location maui windsurfing`

Speicherbild:



Das *wäre* so, als hätte die Shell folgendes Array für uns angelegt:

```
1 char *argv[] = {
2                   "./my-action", "--location",
3                   "maui", "windsurfing"
4                 };
5
6 int argc = sizeof( argv )/sizeof( argv[ 0 ] );
```

„Wieso so zaghaft und wäre, es ist doch auch so!?“ Na ja, nicht ganz. Der Compiler würde die konstanten Zeichenketten wie `"windsurfing"` usw. im Konstanten-Segment anlegen. Aber der Compiler sieht diese Parameter ja gar nicht, sondern erst die Shell. Ergo kann der Compiler auch nichts tun; und die Shell hingegen kann neue Parameter (und andere Werte) nur auf dem Stack ablegen. Daher sind alle diese Zeichenketten keine Konstanten sondern „gewöhnliche“ Daten, die auf dem Stack und nicht im Konstanten-Segment zum Liegen kommen. *„Shit, schon wieder ertappt...“* Macht nichts, du bist ja in einer Lehrveranstaltung und sollst etwas dazulernen.

51.5 Shell und Programm: `_init()` und `exit()`

Bisher haben wir besprochen, wie die Shell (die Kommandoeingabe) ein Programm aufruft und dabei gleich die ganze Kommandozeile als Parameterliste an das Programm übergibt. Nun fehlen noch die beiden Funktionen `_init()`- und `exit()`. Durch diesen Mechanismus gehen nicht nur Informationen von der Shell zum Programm sondern auch vom Programm zurück zur Shell. Wie im Skript öfters erwähnt, wird die `_init()`-Funktion aus der Standardbibliothek (siehe beispielsweise auch die Kapitel 39 und 55) genommen und einfach *immer* dazu gebunden. Diese `_init()`-Funktion hat folgende vier Aufgaben:

1. Erzeugen der beiden Variablen `argc/argv`[1],
2. Initialisierung aller globalen Variablen,
3. Aufruf der `main()`-Funktion und
4. Weiterleiten des Rückgabewertes der `main()`-Funktion an die Shell.

Ein entsprechendes pseudo C-Programm befindet sich oben auf der nächsten Seite. Das Meiste haben wir bereits besprochen. Neu hinzu gekommen sind lediglich die Programmzeilen `10` und `11`. In der Programmzeile `10` wird der Rückgabewert der `main()`-Funktion in einer lokalen Variablen `ret` gespeichert. Dieser Wert wird in Zeile `11` mittels des System Calls `exit()` über das Betriebssystem an die Shell zurück geschickt. Dadurch erfährt die Shell beispielsweise, ob bei der Programmausführung alles gut gegangen oder ein Fehler aufgetreten ist. Diesen Rückgabewert könnt Ihr Euch in der Shell mittels `echo $?` (Linux) bzw. `echo %errorlevel%` (Windows) einfach ansehen. Klingt unglaublich? Einfach mal

[1] Das ist nicht ganz korrekt, denn um das Erzeugen dieses Arrays (und ggf. weiterer Daten wie den Umgebungsvariablen) kümmert sich die Shell und das Betriebssystem gemeinsam. Aber diese Diskussion würde hier zu weit führen, weshalb wir es bei dieser kleinen Ungenauigkeit belassen wollen.

```
1  int _init()
2      {
3          char *argv[] = {
4                              "./my-action", "--location",
5                              "maui", "windsurfing"
6                          };
7          int argc = sizeof( argv )/sizeof( argv[ 0 ] );
8          int ret;
9          // init global variables
10         ret = main( argc, argv );
11         exit( ret );
12     }
```

ausprobieren! Zu erwähnen bleibt noch, dass bei der Shell die Bedeutung des Rückgabewertes anders herum ist: Rückgabewert `0` bedeutet ok, alle anderen Werte bedeuten einen Fehler. Es bleibt noch zu ergänzen, dass nach dem aktuellen C-Standard der C-Compiler automatisch ein `return 0` an das Ende der `main()`-Funktion anhängt.

51.6 Programmargumente und die Shell

Für viele Programmierer, sowohl Anfänger als auch Fortgeschrittene, ist es nicht immer einfach, die einzelnen Funktionen auseinander zu halten. Folgende Regeln sind zu beachten:

1. Der in diesem Kapitel beschriebene `argc/argv`-Mechanismus ist so, wie hier beschrieben. Durch `argc/argv` werden die einzelnen Argumente der Kommandozeile dem Programm zur Verfügung gestellt.
2. Die Kommunikation der Kommandozeilenargumente an das Programm wird durch das Betriebssystem vorgenommen.
3. Die Shell zerlegt die Kommandozeile in ihre einzelnen Bestandteile. Damit liegt es an der Shell, was ein Wort ist und was nicht, worüber wir uns im Rest dieses Abschnitts befassen. Ferner sorgt die Shell dafür, dass das Array `argv` aufgebaut wird.

Beim obigem ersten Überblick haben wir einfach gesagt, dass die einzelnen Wörter durch Leerzeichen und Tabulatoren voneinander getrennt werden. Das ist der Standardfall. *„Aber wie bekommt man Leerzeichen in ein Wort, wenn es doch die Wörter per Definition voneinander trennt?“* Das ist eine gute Frage. Unter Linux gibt es verschiedene Möglichkeiten. Zum einen können wir Zeichenketten in einfache Apostrophs bzw. Anführungszeichen einschließen. Ferner kann man auch den Backslash einsetzen. In diesem Fall gilt das *nächste* und *nur das nächste* Leerzeichen nicht als Worttrenner sondern als Nutzzeichen. Diese Form der Eingabe nennt man auch „Escape“. Statt viel zu reden, präsentieren wir lieber ein paar Beispiele. Dabei notieren wir keine Anführungszeichen bei den Argumenten, um

nicht zu verwirren. Nun aber los:

Kommando	`argv[ 0 ]`	`argv[ 1 ]`	`argv[ 2 ]`	`argv[ 3 ]`
`echo 123 456 789`	`echo`	`123`	`456`	`789`
`echo '123' "456" 789`	`echo`	`123`	`456`	`789`
`echo '123 456' 789`	`echo`	`123 456`	`789`	
`echo "123 456" 789`	`echo`	`123 456`	`789`	
`echo "12"3 4'5'6 789`	`echo`	`123`	`456`	`789`
`echo 123\ 456 789`	`echo`	`123 456`	`789`	
`echo '12"3' "4'5'6" 7\'89`	`echo`	`12"3`	`4'5'6`	`7'89`

Zu diesen Beispielen kommt noch hinzu, dass zumindest in Linux der rückwärts gerichtete Apostroph (Backslash) eine spezielle Bedeutung hat und dass bei den einfachen Apostrophs die Shell-Variablen nicht ersetzt werden. Aber das führt hier viel zu weit. Bei Interesse einfach mal die Betreuer während der Übungszeiten fragen.

„War nicht einfach, aber ich glaube, ich hab's verstanden: Der `argc/argv`*-Mechanismus ist durch den C-Standard festgelegt und eine schöne Methode, auf die gesamte Programmzeile zuzugreifen. Echt elegant. Aber es ist Sache der Shell zu entscheiden, was genau ein Wort ist, wie die einzelnen Wörter getrennt werden und welche Sonderzeichen wie behandelt werden.“* Gut, Du hast es verstanden!

51.7 Ergänzende Anmerkungen

Zu guter letzt müssen wir noch ein paar Anmerkungen ergänzen, auf die wir oben verzichtet haben, um den Blick nicht zu sehr vom Wesentlichen abzulenken.

1. Unter Linux werden sowohl der Apostroph als auch die Anführungszeichen verwendet, in Windows sind nur Anführungszeichen möglich.
2. In einigen Shells kann man sogar einstellen, welche Zeichen tatsächlich als Worttrenner verwendet werden. Dadurch kann eine Shell beispielsweise auch eine Zeile `A:123:xyz` in seine Bestandteile zerlegen. Das ist aber sehr Shell-spezifisch.
3. Nach dem aktuellen C-Standard [13] ist das Array `argv` um einen Null-Zeiger erweitert, der sich *nicht* in der Zahl der durch `argc` vermittelten Argumente wiederfindet. Mit anderen Worten gilt: `argv[ 0.. argc ]` mit `argv[ argc ] == 0`.
4. Bezüglich des Funktionsaufrufs `exit()` bleiben noch zwei Dinge zu erwähnen. Erstens ist es seit einiger Zeit gar kein System Call mehr sondern eine normale Funktion, die den wirklichen System Call namens `_exit()` aufruft und zweitens alle bisher gepufferten Ausgaben auch wirklich zur Ausgabe bringt. Mehr dazu besprechen wir im Skriptteil VI.
5. Durch die `_init()`-Funktion wird das Hauptprogramm immer als `int main( int argc, char **argv )` aufgerufen, egal, wie man `main()` in seinem Programm de-

finiert. Auch ein `main(void)` eurerseits ändert daran nichts. Ebenso erzwingt ein `main(double d)` *nicht*, dass ein double übergeben wird. *Egal,* wie Ihr die `main()`-Funktion deklariert, sie wird *immer* mit den Parametern `argc` und `argv` aufgerufen. Nur, dem Compiler ist egal was Ihr schreibt und es ist ihm zu blöd, einen Fehler zu melden. *Lucky you!*

Kapitel 52

Programmabstürze und sicheres Programmieren

Wenn die Programme komplexer werden, steigt auch naturgemäß sowohl die Zahl der Programmierfehler als auch die Zahl der Programmabstürze. In diesem Kapitel wollen wir herausarbeiten, dass Programmabstürze gar nicht so schlimm sind, sondern auch etwas Gutes haben. *„Also, ich weiß gar nicht, was ihr habt. Meine Programme laufen immer und stürzen nie ab; ich finde es sogar lustig, dass die Programme meiner Kumpels und Freunde so oft abstürzen ;-)“* Vielleicht hast du ja keine Freunde... Ok, Spaß beiseite. In diesem Kapitel besprechen wir erst einmal, warum Programme abstürzen und warum dies sogar gut ist. Anschließend besprechen wir ein paar Beispiele, die zeigen sollen, wie man möglichst sicher programmiert.

52.1 Hintergrund: Ursachen von Programmabstürzen

Eigentlich ist es schon eine merkwürdige Sache, dass ein Programm überhaupt abstürzt. Der Compiler hat doch alles übersetzt. Also können doch gar keine „verbotenen“ Maschineninstruktionen in der ausführbaren Datei vorhanden sein. Also, warum soll es abstürzen? Und in der Tat wird ein Programm selten durch die CPU direkt abgebrochen. Vielmehr sind die meisten Programmabstürze eine Kooperation weiterer Hardwarekomponenten wie beispielsweise der MMU (Memory Management Unit[1]) und dem Betriebssystem. Folgende Problemfälle zählen sicherlich zu den Ursachen der meisten Programmabstürze:

Falsche Adresse: Solange der C-Compiler die benötigten Adressen selbst bestimmt und anschließend auf die entsprechenden Speicherstellen selbst zugreift, kann dies eigentlich nicht zu einem Programmabsturz führen; der Compiler weiß in der Regel, was er

[1] Das Besprechen der MMU würde weit über den Inhalt dieser Lehrveranstaltung hinaus gehen und ist eher Gegenstand von Vorlesungen aus den Bereichen Rechnerarchitekturen, Betriebssysteme und Rechnerorganisation.

macht. Wenn aber der (noch nicht so routinierte) Programmierer mittels Zeigern und Zeigerarithmetik die Adressen selbst bestimmt, können diese falsch sein. Insbesondere kann es sein, dass der Zeiger nicht mehr in den Programmbereich zeigt und entsprechend die Speicherzugriffe außerhalb des Programms sind. Dies wird von der oben angesprochenen Hardware bemerkt, woraufhin das Betriebssystem das Programm mit einer Fehlermeldung abbricht. Ein bekanntes Beispiel hierfür ist der Null-Zeiger:

```
1 int *p = 0;      // der null-zeiger ist noch erlaubt
2 *p = 1;          // der zugriff auf die adresse 0
3                  // fuehrt zum absturz
```

Schreibfehler: In Kapitel 40 haben wir über die Segmente gesprochen, die vom C-Compiler angelegt werden. Dabei sind die unteren beiden Segmente, also das Konstanten-Segment und das Text-Segment, schreibgeschützt. Entsprechend führen Schreibzugriffe auf diese beiden Segmente zu einem Programmabsturz. Entsprechend ist das folgende Beispiele fehlerhaft:

```
1 char *str = "hallo";// str zeigt in das konstanten-segment
2 str[ 0 ] = 'H';     // programmabsturz, da die konstanten
3                     // nicht veraendert werden duerfen
```

Ebenso würde es zu einem Programmabsturz kommen, wenn der Zeiger versehentlich in das Text-Segment zeigen würde, was sich beispielsweise durch die einfache Zuweisung `str = (char *) printf` erreichen ließe; dieser Fehler passiert aber vermutlich nur den wenigsten Programmieranfängern.

Lesefehler: Auch wenn Zeiger in das richtige Segment zeigen, kann es dennoch sein, dass Zugriffe auf die entsprechenden Speicherstellen zu einem Programmabsturz führen. Dies kann daran liegen, dass Vorgaben des Prozessorherstellers nicht eingehalten werden. So schreiben beispielsweise einige Hersteller zwingend vor, dass Zugriffe auf `int` und `double`-Werte immer an geraden Speicheradressen zu erfolgen haben. Sollte es sich aber aufgrund eines Programmierfehlers um eine ungerade Adresse handeln, wird beim Speicherzugriff das Programm abgebrochen.

Unerlaubtes Ausführen: In der Programmiersprache C kann man auch Zeiger auf Funktionen verwenden (siehe hierzu Kapitel 85). Betriebssysteme wie Linux und Windows erwarten aber, dass sich die entsprechenden Funktionen im Text-Segment befinden. Sollte aber ein Funktionszeiger in ein anderes Segment zeigen und das Programm versuchen, dort eine Funktion auszuführen, wird es abgebrochen.

Division durch Null: Im mathematischen Sinne sind Divisionen durch Null nicht besonders sinnvoll. Aber im Falle eines Programmierfehlers kann dies durchaus vorkommen. In Unix/Linux-artigen Betriebssystemen ist es so, dass in C eine ganzzahlige (`int`) Division durch Null zu einem Programmabbruch führt; bei Ausdrücken vom Typ `double` ist dies aber nicht so, es wird nur eine Notiz auf der Console ausgegeben.

Programmabstürze aus obigen Gründen resultieren in der Regel aus Programmierfehlern und sind so oder so ein Ärgernis. Je nach Betriebssystem bekommt der Programmierer einige wenige Hinweise auf die Ursachen. Aber in der Regel erfordert das Auffinden derartiger Fehler einiges an Suchen und Testen.

52.2 Bewertung von Programmabstürzen

Programmabstürze sind ärgerlich, keine Frage, insbesondere weil sich viele (Programmieranfänger) hilflos und peinlich berührt fühlen. Aber letzteres ist völlig unnötig, um hier schon einmal das Ergebnis dieses Abschnittes vorweg zu nehmen.

Wie schon öfters in diesem Skript besprochen, sollte es das Ziel eines jeden Programmierers sein, ein fehlerfreies Programm zu entwickeln. Da fehlerfreie Programme bei *sachgerechter* Verwendung nicht abgebrochen werden, diskutieren wir in diesem Abschnitt nur *fehlerhafte* Programme.

„Also sind meine Programme viel besser als die meiner Freunde, denn meine sind noch nie abgestürzt; ich wusste es schon immer :-))“ Wie viel Freunde hattest/hast du doch gleich? Im Ernst, so einfach ist es nicht. Schauen wir uns doch einfach mal folgendes simples Programm an, das bei mir auf meinem Rechner die Ausgabe `summe= 9526` produziert.

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5         int a[] = { 4711, 815, 2012, 1984 };
 6         int s, i;
 7         for( i = s = 0; i < 5; i++ )
 8             s += a[ i ];
 9         printf( "summe= %d\n", s );
10     }
```

Ein kurzer Blick genügt um festzustellen, dass alles chique ist. *„Ja, ist doch super!“* Genau. Aber ein etwas genauerer Blick zeigt leider, dass das ausgegebene Ergebnis falsch ist, denn die Summe der vier Zahlen beträgt $4711 + 815 + 2012 + 1984 = 9522$ und nicht 9526. Und wenn wir uns das Programm etwas genauer ansehen, dann können wir feststellen, dass in Zeile 7 ein Programmierfehler versteckt ist: es hätte `i < 4` und nicht `i < 5` heißen müssen. Und genau das ist das Problem: Viele Programme, insbesondere größere, beinhalten kleine Fehler[2] die niemand bemerkt. In diesem Beispiel wäre es geradezu famos, wenn das Programm abgebrochen würde! Denn dann wüsste der Programmierer, dass er in Zeile 8

[2] Wer wissen möchte, warum das Programm nicht abbricht und weshalb das Ergebnis zustande kommt, der gebe sich die Adressen aller Variablen mittels `printf( "%p", ... );` aus, male ein Speicherbildchen und führe eine Handsimulation durch.

auf ein Element zugreift, dass gar nicht existiert. In anderen Programmiersprachen, wie beispielsweise Pascal, kann man den Compiler sogar so einstellen, dass sich das Programm in solchen Fällen mit einer sehr aussagekräftigen Fehlermeldung beendet.

Auch wenn wir obige Diskussion nur an einem sehr einfachen Beispiel geführt haben, so handelt es sich dennoch um ein generelles Problem. Es gibt genügend viele wissenschaftliche Veröffentlichungen, Diplomarbeiten und sogar Doktorarbeiten, deren Resultate auf Programmierfehlern beruhen aber von den Autoren und vielen anderen nie bemerkt wurden. Insofern ist es besonders gut, wenn fehlerhafte Programme nicht einfach weiter laufen sondern abbrechen, denn sie geben dem Programmierer wertvolle Hinweise über mögliche Fehler. *„Ok, überredet, ich schaue mir mal meine Programme nochmals an. . .“*

52.3 Weiterführende Maßnahmen

Natürlich ist es schön, wenn ein Programm nicht abstürzt. Neben den im ersten Abschnitt diskutierten Gründen liegt eine weitere Ursache in der nicht intendierten Verwendung von Programmen. *„Bitte was? Nicht intendiert?“* Ja, das bedeutet, dass ein Programm anders verwendet wird, als es sich der Programmierer ursprünglich gedacht hat. *Na, wenn der Nutzer so doof ist, kann ich doch auch nichts dafür!“* Mag man meinen, aber in den Übungen beobachten wir oft, dass Programmierer und Nutzer ein und dieselbe Person sind. *„Oops!“*

Selbst wenn man als Programmierer alles im Griff hat und ein fehlerfreies Programm erstellt, kann man sich nicht sicher sein, dass der Nutzer alles richtig verwendet. Daher sollte man die Eingabeschnittstelle (hin zum Nutzer) durch Einbau verschiedener Abfragen möglichst sicher machen. Aber schauen wir uns doch einfach mal ein paar Beispiele an, bevor wir hier lange herumphilosophieren.

Absicherung von `argc/argv`: Sehen wir uns folgendes Beispiel an, das die Summe zweier Zahlen berechnet, die als Argumente übergeben werden (die Funktion `atoi()` wandelt die übergebene Zeichenkette in die entsprechende Zahl um):

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main( int argc, char **argv )
5     {
6        int arg1 = atoi( argv[ 1 ] );
7        int arg2 = atoi( argv[ 2 ] );
8        printf( "summe= %d\n", arg1 + arg2 );
9     }
```

Beim *beabsichtigten* Aufruf `./test 12 34` wird auch tatsächlich 46 ausgegeben. Was passiert nun aber, wenn der Nutzer einen der beiden oder beide Parameter vergisst? Dann wird das Programm mit der Fehlermeldung `Segmentation fault` abgebrochen. Warum

passiert das? In diesem Fall werden Null-Zeiger an die Funktion `atoi()` übergeben, deren Dereferenzierung zum Programmabsturz führt. Daher ist es sinnvoll, den Programmanfang so zu gestalten, dass bei nicht sachgerechter Verwendung das Programm mit einer selbst generierten Fehlermeldung kontrolliert abbricht:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main( int argc, char **argv )
5      {
6         int arg1, arg2;
7         if ( argc != 3 )
8         {
9            printf( "Sorry, falsche Zahl von Parametern\n" );
10           printf( "Korrekt: %s Zahl_1 Zahl_2\n", argv[ 0 ] );
11           exit( 1 );
12        }
13        arg1 = atoi( argv[ 1 ] );
14        arg2 = atoi( argv[ 2 ] );
15        printf( "summe= %d\n", arg1 + arg2 );
16     }
```

Selbstschutz: Die eben beschriebene Herangehensweise kann man bei Programmen mit steigender Komplexität (mit vielleicht mehr als zehn selbst geschriebenen Funktionen) für sich selbst übernehmen, um sich vor sich selbst zu schützen. Nehmen wir als Beispiel eine eigene Funktion `myStrlen()`, die die Länge einer Zeichenkette ermittelt:

```
1  int myStrlen( char *str )
2      {
3         int len;
4         for( len = 0; *str != '\0'; str++ )
5            len++;
6         return len;
7      }
```

Was passiert, wenn wir diese Funktion (versehentlich) mit einem Null-Zeiger aufrufen? Richtig, das Programm wird vom Betriebssystem abgebrochen und wir müssen uns an die Fehlersuche machen. In größeren Projekten wäre es zumindest während der Entwicklung sinnvoll, diese Funktion gegenüber ungewollten Aufrufen abzusichern:

```
1  int myStrlen( char *str )
2      {
3         int len;
4         if ( str == 0 )
5         {
6            printf( "myStrlen: sorry, Null-Zeiger! " );
7            printf( "See you later, Aligator\n" );
8            exit( 1 );
9         }
10        for( len = 0; *str != '\0'; str++ )
11           len++;
12        return len;
13     }
```

Fortgeschrittene Programmierer würden diese „Kontrollanweisungen“ in der Regel durch entsprechende Präprozessordirektiven einklammern, sodass diese Anweisungen in der „produktiven“ Software nicht mehr enthalten sind:

```
1  int myStrlen( char *str )
2      {
3         int len;
4         #ifdef MY_DEBUG
5         if ( str == 0 )
6         {
7            printf( "myStrlen: sorry, Null-Zeiger!  " );
8            printf( "See you later, Aligator\n" );
9            exit( 1 );
10        }
11        #endif
12        for( len = 0; *str != '\0'; str++ )
13           len++;
14        return len;
15     }
```

Aber eine weitergehende Diskussion würde jetzt hier zu weit führen.

Kapitel 53

Zusammengesetzte Datentypen: `struct`

Im bisherigen Skript haben wir einfache Datentypen (`int`, `char` und `double`) sowie ein- und mehrdimensionale Arrays und Zeiger behandelt. Ein gemeinsames Charakteristikum dieser Datentypen ist, dass entsprechende Variablen immer nur einen oder mehrere Werte vom selben Typ aufnehmen können; das gleichzeitige Verwenden mehrerer Werte *unterschiedlicher* Typen war nicht möglich. Prinzipiell ist dies auch gar nicht notwendig, denn man kann beliebig viele Variablen definieren, um alle Daten zu verwalten. Durch eine geschickte Wahl der Namen kann man andeuten, dass diese Variablen etwas miteinander zu tun haben. Allerdings ist es gestaltungstechnisch sowie aus Sicht der Änderungsfreundlichkeit wünschenswert, unterschiedliche Datentypen zu einem komplexen Datentyp zusammenzufassen. Hierfür werden von der Programmiersprache C `struct`s angeboten, die wir in diesem Kapitel behandeln.

53.1 Problemstellung

Nehmen wir an, wir wollten ein Programm zur Verwaltung von Windsurfbrettern entwickeln. Dann bräuchten wir für jedes Windsurfbrett folgende Angaben: Länge in cm, Volumen in Liter (mit Nachkommastellen) und eine Bezeichnung. Das wären drei unterschiedliche Angaben mit den Datentypen `int`, `double` und `char *`. Wenn wir nun zehn Bretter verwalten wollten, bräuchten wir drei Arrays mit jeweils zehn Elementen. Und schon wird es unhandlich. Wollten wir beispielsweise zwei Einträge vertauschen, müssten wir dies in drei Arrays machen. Wollten wir etwas hinzufügen, z.B. den Namen des Herstellers, müssten wir sicherlich mehrere Stellen im Programm ändern. Dies alles ist weder übersichtlich, noch änderungsfreundlich und vor allem fehleranfällig. Schön wäre es, wenn wir alles zusammenpacken könnten.

53.2 Verwendung Datentyp `struct`

Ein `struct` kann mehrere Komponenten, die jeweils von unterschiedlichem Datentyp sein *können,* zu einem neuen, komplexen bzw. strukturierten Datentyp zusammenfassen. Dieser Datentyp kann anschließend wie jeder andere Datentyp verwendet werden.

C-Syntax

```
struct board {
   int length;
   double volume;
   char *id;
};
struct board b1, b2;
```

Abstrakte Programmierung

```
Datentyp board
  Integer: length
  Double : volume
  Zeichenkette: id

Variablen: Typ board: b1, b2
```

Hinweise: Bei der Verwendung von `struct`s sollten folgende Dinge beachtet werden:

1. Die einzelnen Komponenten eines `struct` können beliebigen Typs sein. Dies schließt andere `struct`s und Arrays ein. Allerdings müssen diese Datentypen bereits *vorher* definiert worden sein. Dies bedeutet, dass sich ein `struct` *nicht* selbst enthalten kann, denn es ist erst nach Abschluss seiner Definition bekannt. Desweiteren kann man unterschiedliche `struct`s nicht „über kreuz“ ineinander einfügen; einer der beiden `struct`s kann erst nach dem anderen definiert und kann somit nicht schon voher verwendet werden.

2. Auf die einzelnen Komponenten greift man mittels des Punktes `.` zu. Beispiele: `b1.length = 234` und `b2.volume = 85.4`.

3. Wenn innerhalb eines `struct`s ein weiterer `struct` ist, muss der Punkt-Selektor entsprechend oft angegeben werden:

```
 1 struct s1 { int i; char c; };
 2
 3 struct s2 { double d; struct s1 s; };
 4
 5 int main( int argc, char **argv )
 6     {
 7         struct s2 data;
 8
 9         data.d = 123.456;
10         data.s.i = 4711;
11         data.s.c = 'X';
12     }
```

4. `struct`s sind zuweisungskompatibel. D.h., zwei Variablen vom selben `struct` können einander zugewiesen werden. Beispiel: `b2 = b1`. In diesem Fall werden *alle* Komponenten von `b1` an `b2` zugewiesen.

5. Da `structs` zuweisungskompatibel sind, können sie auch als Rückgabewert einer Funktion erscheinen. Beispiel:

```
1  struct board f(){ struct board b; ... return b; }
2
3  int main( int argc, char **argv )
4      {
5          struct board my_board;
6          my_board = f();
7      }
```

   In Zeile 6 wird durch die Funktion `f()` ein ganzes `struct board` zurückgegeben, das der Variablen `my_board` zugewiesen wird.

6. Durch die Einbettung in ein `struct` können auch Arrays zuweisungskompatibel gemacht werden (da sie eine definierte, unveränderbare Länge haben). Beispiel:

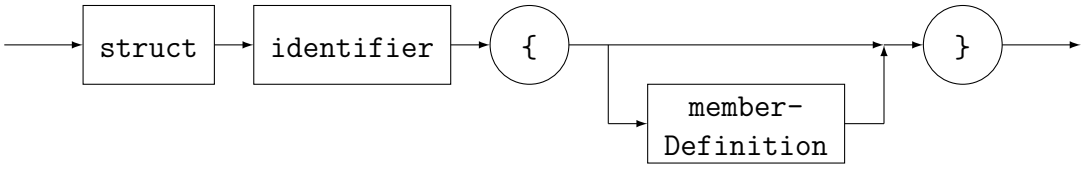
```
1  struct arr_10 { int array[ 10 ]; }
2
3  int main( int argc, char **argv )
4      {
5          struct arr_10 a, b;
6          int i;
7          for( i = 0; i < 10; i++ )
8              a.array[ i ] = 4711;
9          b = a;
10     }
```

   Am Ende von Zeile 9 haben auch alle Komponenten des Arrays `array` der Variablen `b` den Wert 4711.

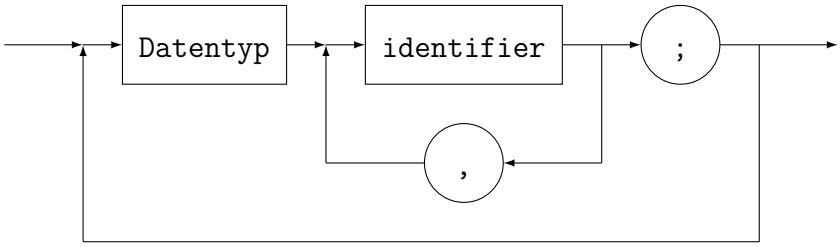
7. Der Adressoperator `&` läßt sich selbstverständlich sowohl auf `structs` als auch die einzelnen Komponenten anwenden. Beispiele: `& my_board`, `& my_board.length` und `& my_board.volume`.

53.3 Syntaxdiagramme

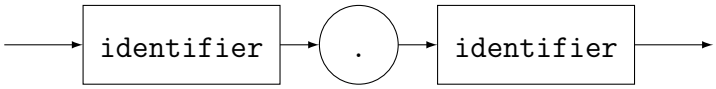
struct



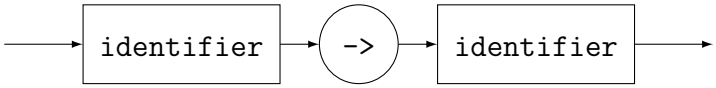
member-Definition



member-Zugriff



member-Zeiger-Zugriff



53.4 Korrekte Beispiele

Siehe oben und unten in diesem Kapitel.

53.5 Fehlerhafte Beispiele

Fehler bei der Verwendung von `structs` beziehen sich meist auf vergessene Klammern und/oder Punkte als Selektoren.

53.6 Ausgabe von `structs`

Datensätze, die `structs` sind, können erwartungsgemäß nicht direkt als Ganzes ausgegeben werden; sie müssen wie alle anderen strukturierten Datentypen elementweise ausgegeben werden. Für die Strukturierung des eigenen Programms ist es meistens recht sinnvoll, hierfür eine eigene Ausgabefunktion zu implementieren.

53.7 Definition einschließlich Initialisierung

Wie bei den anderen Datentypen auch, kann man `struct`-Variablen bei ihrer Definition auch gleichzeitig initialisieren. Beispiel: `struct board my_board = {234, 105.0, "JP"}`

Bei gleichzeitiger Initialisierung ganzer Arrays bietet sich die Strukturierung mittels geschweifter Klammern an, wie wir es auch bei mehrdimensionalen Arrays gesehen haben (siehe auch Kapitel 49).

```
1  struct board {
2                  int length;
3                  double volume;
4                  char *id;
5                };
6
7  int main( int argc, char **argv )
8      {
9         struct board my_boards[ 2 ] =
10                      {
11                        { 240, 110.0, "JP" },
12                        { 220, 105.0, "simmer style" }
13                      };
14     }
```

53.8 Zeiger auf `structs`: der `->` Operator

Da `structs` ganz normale Datentypen sind, kann man auch Zeiger auf selbige definieren. Bezugnehmend auf obiges Beispiel könnte man schreiben: `struct board *ptr`. Auf die einzelnen Komponenten kann man wie folgt zugreifen:

```
1  struct board mb, *ptr;  // ggf. mittels p=& mp initialisieren
2
3  mb.length = 234;         (*ptr).length = 234;
4  mb.id = "simmer style"; (*ptr).id = "simmer style";
```

Beide Fällen zeigen, dass die Struktur „Variable.Komponente“ erhalten geblieben ist. Nur ist die Notation „`(*ptr).`“ auf Dauer etwas mühsam. Zur Vereinfachung gibt es den `->` Operator:

```
1  struct board mb, *ptr;  // ggf. mittels p=& mp initialisieren
2
3  (*ptr).length = 234;         ptr->length = 234;
4  (*ptr).volume = 85.45;       ptr->volume = 85.45;
5  (*ptr).id = "simmer style"; ptr->id = "simmer style";
```

In diesem Beispiel sind jeweils die rechte und linke Seite identisch.

Kapitel 54

`typedef`: Selbstdefinierte Datentypen

Schon wieder Datentypen ... Reicht es nicht langsam? Na ja, noch nicht. Die grundlegenden Datentypen wie `char`, `int` und `double` (siehe bei Interesse aber auch noch Skriptteil VIII) hatten wir bereits. Hinzu kamen noch Arrays, `structs` und Zeiger. Wie steht es nun mit der Änderungsfreundlichkeit, der Wartbarkeit und dem Schreibaufwand? Geht so, wäre die Antwort, wie wir gleich sehen werden. Um die daraus resultierenden Probleme etwas zu lindern, stellt die Programmiersprache das Konzept `typedef` zur Verfügung, mit dem man seine eigenen Datentypen einigermaßen änderungsfreundlich definieren kann. Aber wie immer benötigen wir wieder ein wenig Übung, um auch dieses Konzept erfolgreich anwenden zu können.

54.1 Aspekt: Änderungsfreundlichkeit/Wartbarkeit

Stellen wir uns vor, wir haben ein längeres Programm, in dem wir an vielen Stellen eine ID (z.B. für ein Auto) verwalten müssen. Und am Anfang unseres Entwurfs haben wir uns mit unseren Freunden darauf geeinigt, dass die ID eine Variable vom Typ `int` sein sollte. Ist nämlich praktisch und damit kennen wir uns aus.

Nun stellen wir uns vor, dass wir aus irgendeinem Grund merken, dass `int` gar nicht so geeignet ist und eigentlich `char *` viel besser passen würde. Lösung 1: Das Problem wird ignoriert und man versucht krampfhaft drum herum zu arbeiten. Erfolgsaussichten? Auf lange Sicht sehr gering. Lösung 2: Wir gehen die 589 Zeilen unseres C-Programms durch und ändern fleißig. Erfolgsaussichten? Mäßig, viel Arbeit und fehleranfällig. Warum? Nun, hier und da werden wir eine Änderung vergessen oder einmal zu viel anpassen.

Wunschlösung: Wir haben einen Mechnanismus, der uns die Arbeit einfach macht: Eine Änderung an einer Stelle und die meiste Arbeit ist erledigt.

54.2 Aspekt: Notation von Zeiger-Typen

Mit jedem zweiten C-Programmierer gibt es über kurz oder lang eine Diskussion darüber, wie man denn nun Zeiger richtig definieren sollte. Die beiden möglichen Varianten sind: `(1) int* i_ptr;` und `(2) int *i_ptr;`.

Dem Compiler ist es egal, die Mehrheit der C-Programmierer votiert üblicherweise für die erste Variante. Argument: Der Stern gehört zum Datentyp und sollte daher am Basistyp und nicht an der Variablen stehen. Klingt plausibel, ist aber nicht haltbar. Sehen wir uns folgende Deklaration an:

```
1 int* p1, p2;
```

Sieht gut aus, macht aber nicht das, was viele erwarten und die Fraktion Variante (1) suggeriert: Sollte es sich bei `int*` tatsächlich um einen Typ handeln, wären `p1` und `p2` beides Zeiger auf ein `int`. Aber wie Ihr alle wisst, ist das nicht so, `p2` ist kein Zeiger sondern eine gewöhnliche Variable vom Typ `int`. Mit anderen Worten, die Notation `int* p1, p2` ist zumindest kontraintuitiv. In C ist es nun mal so. Und C ist nicht Pascal, dort wäre es schöner!

54.3 Lösungsmöglichkeit: `typedef`

Mit `typedef` kann man tatsächlich einen *neuen* Typ definieren. Beispielsweise könnten wir schreiben: `typedef char *ID_TYPE`. Diesen neunen Typ `ID_TYPE` könnten wir überall im Programm verwenden, als wäre er schon immer da gewesen. Beispiel:

```
 1 typedef char *ID_TYPE;
 2
 3 int prt_id( ID_TYPE id )
 4     {
 5        // ein paar Zeilen zum Drucken der id
 6     }
 7
 8 int main( int argc, char **argv )
 9     {
10        ID_TYPE my_id = ... ;
11        prt_id( my_id );
12     }
```

Eine Änderung des Datentyps ist schnell gemacht: Zeile 1 anpassen (z.B. in ein `struct` umwandeln) und schon geht es weiter. Natürlich muss man ggf. die eine oder andere Funktion anpassen. Aber das sind klar definierte Orte, die man in einem gut strukturierten Programm gut findet.

Und bei den Zeigern klappt es genauso:

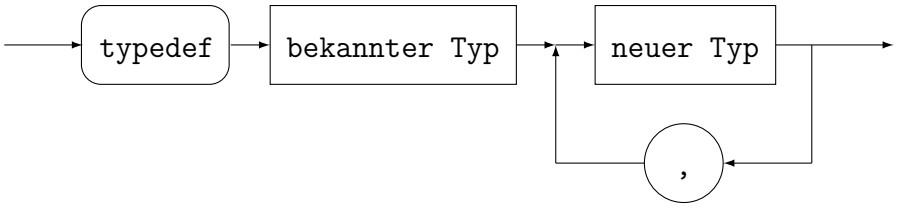
```
 1 typedef int *INT_PTR;           // ein Zeiger auf ein int
 2
 3 int swap_IP( INT_PTR *p1, INT_PTR *p2 )
 4     {
 5        INT_PTR help;
 6        help = *p1; *p1 = *p2; *p2 = help;
 7        return 1;
 8     }
 9
10 int main( int argc, char **argv )
11     {
12        INT_PTR p1, p2;
13        // ein bisschen Code
14        swap_IP( & p1, & p2 );
15     }
```

In Zeile 1 haben wir einen neuen Typ namens `INT_PTR`, der ein Zeiger auf ein `int` ist. In Zeile 12 haben wir gleich zwei Variablen davon deklariert, nämlich `p1` und `p2`. Beide Variablen sind nun Zeiger, als wenn wir `int *p1, *p2` geschrieben hätten. In den Zeilen 3 bis 8 haben wir eine Funktion `swap_IP()`, die nicht zwei `int` sondern zwei Zeiger auf `int` vertauscht. Und obwohl es sich innerhalb von `swap_IP()` um „Doppelzeiger“ handelt, sieht man dies nicht direkt; die Funktion sieht von der Struktur aus als würde sie zwei normale Variablen vertauschen.

54.4 Syntaxdiagramm

Die `typedef`-Syntax ist sehr einfach:

typedef



Ausgehend von einem bekannten Datentyp können einfach neue definiert werden, die ihrerseits wie alle bereits bekannten Typen verwendet werden können. Der nächste Abschnitt zeigt ein paar einfache Beispiele.

54.5 Korrekte Beispiele

```
 1 typedef int *IP, **IPP;      // int pointer, int point pointer
 2 typedef struct {
 3                   double real;                 // Realteil
 4                   double imaginaer;            // Imaginaerteil
 5                 } my_complex, *my_complex_ptr; // Struct/Zeiger
 6
 7 int main( int argc, char **argv )
 8     {
 9        my_complex z1, z2;       // zwei komplexe Zahlen
10        my_complex_ptr cp1;      // und ein Zeiger
11
12        z1.real      = 10.3;
13        z2.imaginaer = -3.0;
14        cp1          = & z2;
15        cp1->real    = -3.14;
16     }
```

Anmerkung: In den Zeilen 2 bis 5 wird ein Datentyp für komplexe Zahlen definiert. Gleichzeitig wird auch noch ein Zeiger auf diesen Datentyp definiert, was nicht jeder mag. Eine mögliche Alternative besteht darin, im ersten `typedef` nur den Datentyp `my_complex` zu definieren und anschließend zu schreiben: `typedef my_complex *my_complex_ptr`, denn beim zweiten `typedef` ist `my_complex` bereits bekannt, sodass dieser Datentyp ebenfalls verwendet werden darf.

54.6 Fehlerhafte Beispiele

Die Fehlermöglichkeiten sind nicht sehr groß. Meistens fehlt ein Komma, ein Semikolon oder ein verwendeter Datentyp ist noch nicht definiert.

Kapitel 55

„Module“ und getrenntes Übersetzen

Mit besser werdenden Programmierfertigkeiten werden auch die Programme und Projekte immer größer. Über kurz oder lang wird es immer schwerer, den Überblick über eine Datei zu behalten, die 2000 Zeilen oder mehr lang ist. Ferner muss immer alles vom Compiler übersetzt werden, auch wenn man nur eine klitzekleine Änderung vorgenommen hat. Aber zum Glück kann man in C seinen Quelltext auf mehrere Dateien aufteilen und mit Hilfe des Compilers (vor allem des Linkers) zu einem Programm zusammenfügen. Ein weiterer Vorteil dieses getrennten Übersetzens ist, dass mehrere Leute gleichzeitig mitarbeiten und damit die Arbeit beschleunigen können. Allerdings muss man auch einige Fallstricke beachten, damit am Ende nichts schief geht. Und da wir an einer Universität sind, besprechen wir auch, wie dies alles funktioniert.

55.1 Wiederholung

Zum Einstieg wiederholen wir die wichtigsten Arbeitsschritte des Compilers, wie wir sie bereits in Kapitel 39 besprochen haben. Für dieses Kapitel ist Kapitel 40 ebenfalls wichtig, da wir dort die einzelnen Speichersegmente erläutert haben.

Frage 1: Was ist das Ergebnis nach dem Übersetzen und vor dem Linken?

> Am Ende des Übersetzungsvorgangs haben wir den (gesamten) Maschinencode, den der Compiler üblicherweise in eine `.o`-Datei schreibt. In dieser Datei fehlen noch einige Bibliotheken, die noch zusätzlich dazugebunden werden müssen.
>
> An dieser Stelle ist wichtig, zu verstehen, dass diese `.o`-Dateien auch dann erstellt werden, wenn man vom Compiler beispielsweise mittels `gcc -o uebung1 uebung1.c` alles in einem Arbeitsgang machen lässt. Der Compiler erstellt die `.o`-Dateien wie üblich, doch löscht er sie am Ende des Übersetzungsvorgans wieder, sodass wir sie nicht sehen können.

Frage 2: Sind die `.o`-Dateien bereits fertige, lauffähige Programme?

Nein, sind sie nicht. Ihnen fehlen noch die Bibliotheksfunktionen wie die Ein- und Ausgabe und ein paar weitere Kleinigkeiten wie zum Beispiel die `_init()`-Funktion.

Frage 3: Erzeugt der Compiler auch ein Stack Segment?

Definitiv nein. Er erzeugt zwar die Stack Frames. Aber diese werden erst bei den konkreten Aufrufen der Funktionen im Arbeitsspeicher angelegt. Der Compiler erzeugt „nur“ das Text/Code Segment, das Konstanten-Segment und das Data-Segment (siehe auch Kapitel 40). Der Compiler erzeugt auch kein BSS-Segment; er bestimmt nur dessen Größe, angelegt wird es erst beim Programmstart.

55.2 Erläuterung am Beispiel

Nehmen wir einfach mal an, wir haben folgendes, äußerst komplexes Programm:

```
 1 #include <stdio.h>
 2
 3 int hello( char *name )
 4     {
 5         printf( "hello, %s is speaking\n", name );
 6         return 1;
 7     }
 8
 9 int main( int argc, char **argv )
10     {
11         hello( argv[ 0 ] );
12         return 0;
13     }
```

Es sollte einfach zu verstehen sein. In Zeile 11 wird die Funktion `hello( argv[ 0 ])` aufgerufen, der als Parameter der Programmname `argv[ 0 ]` mitgegeben wird. Und die Funktion `hello()` gibt neben diesem Namen noch ein bisschen sinnfreien Text aus.

Die üblichen Compiler-Aufrufe wären jetzt:

	Kommando	Funktion	Eingabe	Ausgabe
1.	`gcc -o beispiel beispiel.c` `./beispiel`	Alles auf einmal Programmaufruf	`beispiel.c`	`beispiel`

2.	`gcc -c beispiel.c`
	`gcc -o beispiel beispiel.o`
	`./beispiel`
		Erst übersetzen
		dann Binden
		Programmaufruf
			`beispiel.c`
			`beispiel.o`
				`beispiel.o`
				`beispiel`

Nun stellen wir uns einfach vor, wir würden jede Funktion einzeln in eine gesonderte Datei packen (siehe unbedingt auch die weiteren Ausführungen):

main.c

```
1  int main( int argc, char **argv )
2      {
3          hello( argv[ 0 ] );
4          return 0;
5      }
```

hello.c

```
1  #include <stdio.h>
2
3  int hello( char *name )
4      {
5          printf( "hello, %s is speaking\n", name );
6          return 1;
7      }
```

Nun müssten wir die Einzelteile übersetzen und anschließend binden (linken). Hierfür hätten wir folgende Möglichkeiten:

	Kommando	Eingabe	Ausgabe
1.	`gcc -o hello main.c hello.c` `./hello`	`main.c`, `hello.c`	`hello`

2.	`gcc -c main.c`
	`gcc -c hello.c`
	`gcc -o hello main.o hello.o`
	`./hello`
		`main.c`
		`hello.c`
		`main.o`, `hello.o`
			`main.o`
			`hello.o`
			`hello`

3.	`gcc -c hello.c`
	`gcc -o hello main.c hello.o`
	`./hello`
		`hello.c`
		`main.c`, `hello.o`
			`hello.o`
			`hello`

Im ersten Beispiel werden wie oben beide Dateien an den Compiler übergeben, übersetzt und sogleich zu einem ausführbaren Programm zusammengebunden. Insofern besteht kein Unterschied zum bisherigen Vorgehen.

Im zweiten Beispiel werden beide Dateien zunächst einmal separat übersetzt und in den entsprechenden `.o`-Dateien zwischengespeichert. Erst anschließend wird daraus ein lauffähiges Programm gemacht. Diese Variante ist besonders vorteilhaft, denn bei Änderungen braucht man nur diejenige(n) Datei(en) zu übersetzen, an denen man etwas geändert hat.

Die dritte Variante ist ein Gemisch aus den ersten beiden Varianten. Sie soll nur die prinzipiellen Möglichkeiten aufzeigen; irgendwelche Vorteile gegenüber den anderen beiden Varianten bietet sie nicht.

55.3 Konsistenz mittels Header-Dateien

Besonders „clever“ ist obige Lösung noch nicht, denn sie lädt geradzu ein, Fehler zu machen. Sollten wir jetzt aus irgendeinem unerfindlichen Grund die Implementierung der Funktion `hello()` in beispielsweise `int hello(int time, char *name)` ändern, ohne auch den Aufruf in `main()` anzupassen, kommt Murks heraus!

„Aber warum kommt Murks heraus? Ich kann doch beide Dateien neu übersetzen. Dann merkt doch der Compiler, dass was faul ist.“ Nein, merkt er nicht. Der Grund hierfür ist ganz einfach: Der Compiler übersetzt Datei für Datei, immer schön nacheinander, auch wenn wir mehrere Dateien auf einmal angeben. Und am Ende *jeder* Datei, wenn er das Ergebnis in die entsprechende `.o`-Datei geschrieben hat, vergisst er sofort *alles*[1]! Entsprechend weiß der Compiler auch nicht, wie die Funktion `hello()` implementiert ist, wenn er sie aufruft und umgekehrt. Genau hier kommen die Header-Dateien (siehe auch Kapitel 38) sowie die Funktions*deklarationen* zum Tragen, die wir bereits in Kapitel 44 erwähnt haben. Wir würden also eine Datei `hello.h` mit dem entsprechenden *Funktionsprototypen* erstellen und diese Datei in beide `.c`-Dateien einbinden:

hello.h

```
1 int hello( char *name );        // function prototype
```

main.c

```
1 #include "hello.h"
2
3 int main( int argc, char **argv )
4     {
5         hello( argv[ 0 ] );
6         return 0;
7     }
```

hello.c

```
1 #include <stdio.h>
2 #include "hello.h"
3
4 int hello( char *name )
5     {
6         printf( "hello, %s is speaking\n", name );
7         return 1;
8     }
```

Würden wir jetzt auch nur eine Stelle ändern, würde uns der Compiler Bescheid geben, da die Prototypen nicht mehr passen.

[1] Warum bloß fallen uns jetzt gerade Schüler und Studenten unmittelbar nach ihren Prüfungen ein ;-) Keine Sorge, wir waren auch mal Schüler und Studenten und haben alles selbst erlebt ...

55.4 Getrenntes Übersetzen: technische Umsetzung

Wie eingangs angedroht, beschreiben wir hier in groben Zügen, wie das getrennte Übersetzen und das anschließende Binden in etwa vonstatten geht. Auf der einen Seite ist diese Diskussion ein wenig akademisch. Aber auf der anderen Seite benötigen wir die wesentlichen Konzepte auch im nächsten Kapitel. Das nachfolgende Bild versucht, den recht komplexen Sachverhalt zu illustrieren, den wir gemäß Eures Wissensstandes deutlich vereinfachen aber im Kern der Sache doch korrekt darstellen.

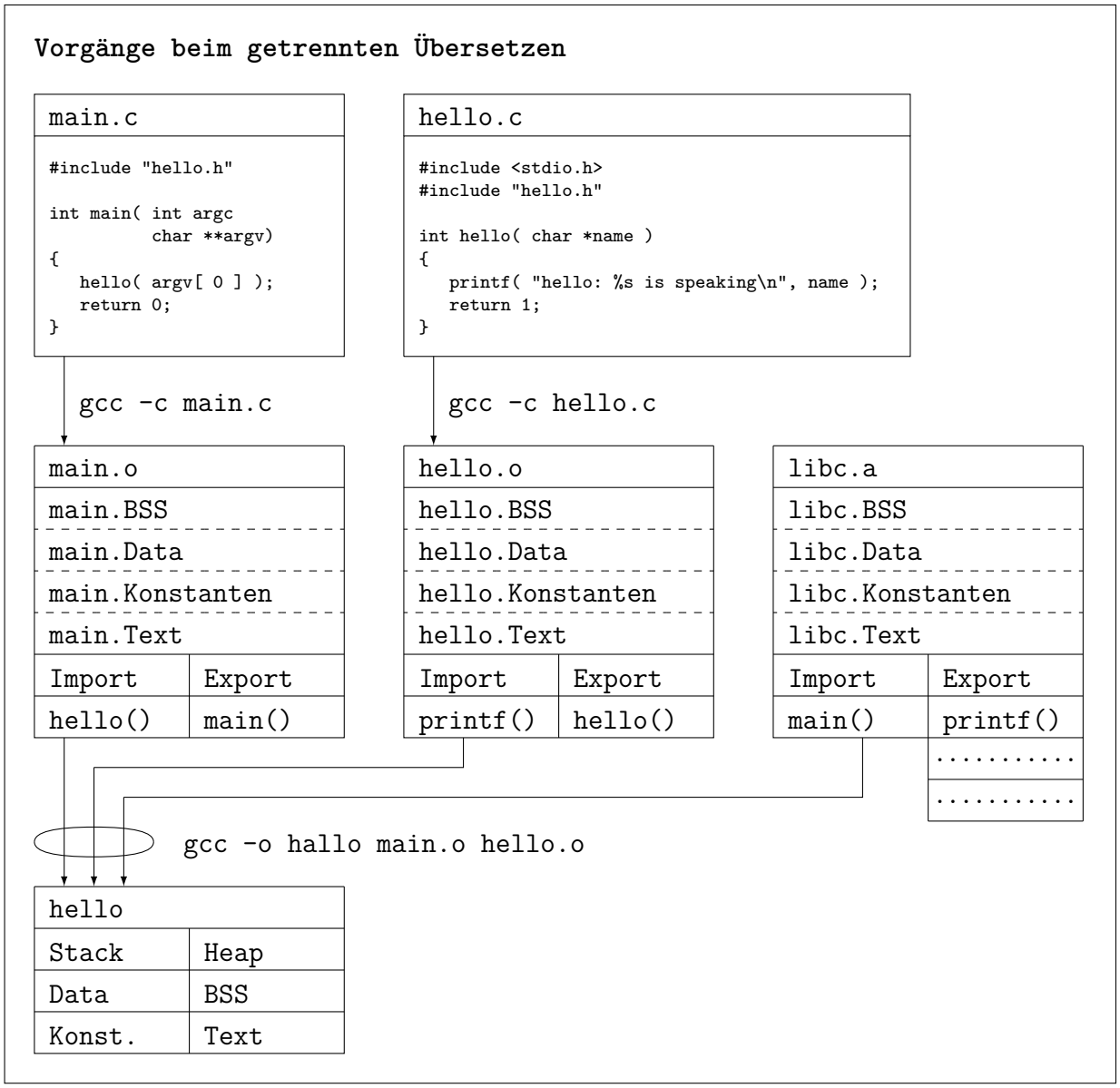
Die Grafik auf der nächsten Seite zeigt noch einmal, dass die einzelnen Dateien durch das Kommando `gcc -c main.c` bzw. `gcc -c hello.c` übersetzt weden. Dabei entstehen zwei neue Dateien `main.o` bzw. `hello.o`, die jeweils aus vier Segmenten sowie je einer Export- und einer Importliste bestehen. Jede dieser Exportlisten gibt an, welche „Objekte“ (Funktionen bzw. Variablen) diese Datei implementiert und damit für „andere“ Dateien zur Verfügung stellen. Im Gegensatz dazu geben die Importlisten an, welche Objekte in den jeweiligen Dateien verwendet aber nicht selbst implementiert sind. Beispielsweise sehen wir, dass die Datei `main.o` ein Objekt `hello()` importiert, dafür aber das Objekt `main()` exportiert.

Der Linker, aufgerufen durch `gcc -o hello main.o hello.o`, versucht nun alle Dateien zusammenzufügen und bindet dazu *auf jeden Fall* die C-Standardbibliothek `libc` (bzw. `glibc`) dazu. Zu diesem Zweck kreiert er zunächst die Datei für das lauffähige Programm (`hello`) und packt alle gleichen Segmente ensprechend zusammen. Anschließend arbeitet er alle Import-/Exportlisten ab und ersetzt die Platzhalter durch die konkreten Anfangsadressen, die die Objekte in den jeweiligen Segmenten später im Arbeitsspeicher haben werden.

Das Auflösen dieser Import- und Exportlisten ist an eine entscheidende Randbedingungen geknüpft: Für jeden Import muss es einen, und zwar genau einen, Export geben. Sollte der Linker für einen Import keinen Export finden, gibt es eine Fehlermeldung der Art `Unresolved external` oder `undefined reference to`. Für den Fall, dass es mehr als einen gleichlautenden Export gibt, erscheint als Fehlermeldung `multiple definition of`. Ein Blick auf das Bild zeigt uns, dass alle Importe und Exporte eindeutig aufgelöst werden können.

Eine letzte Bemerkung betrifft noch die Typüberprüfung beim Binden. Um es kristallklar zu sagen: Weder hat der Linker irgendwelche Typinformation noch führt er irgendwelche Überprüfungen durch! *„Und was soll mir das sagen?“* Zwei Dinge: Erstens muss man sorgfältig Programmieren und zweitens können Fehler entstehen, die man nur schwer findet. *„Vielleicht ein Beispiel?“* Klar, warum nicht. Folgendes Programm wird völlig korrekt ohne jegliche Fehlermeldung oder Warnung übersetzt, stürzt aber sofort ab:

```
1 int main;
```



„Was soll der Quatsch?“ Ganz einfach. Das Objekt `main` ist zwar „nur“ eine Variable vom Typ `int`, erscheint aber dennoch *ohne* jegliche Typinformation in der Exportliste. Der Linker sucht ein Objekt names `main`, da es in der Importliste der Bibliothek `libc` steht und bindet die Dinge fleißig zusammen, obwohl sie nicht passen. Und beim Programmstart wird zwar das Objekt `main()` angesprungen, doch merkt erst das Betriebssystem, dass dort gar keine Funktion sondern eine Variable steht. Tja, so kann’s gehen.

Kapitel 56

Variablen: Sichtbarkeit und Lebensdauer

Sichtbarkeit (auch Gültigkeitsbereich oder engl. Scope genannt) und Lebensdauer sind zwei völlig unterschiedliche Dinge. Ersteres bezieht sich darauf, was man wo verwenden kann. Zweiteres sagt etwas darüber aus, wie lange die Objekte leben. Bei beiden Konzepten sind ein paar Kleinigkeiten zu beachten, die wir in diesem Kapitel erläutern werden. Kleine Schwierigkeiten entstehen vor allem dadurch, dass C-Programme auf mehrere Dateien verteilt werden können. Für die weitere Diskussion sind insbesondere die Kapitel 40 und 55 von großer Wichtigkeit. Also, im Zweifel nochmals anschauen.

56.1 Regelkunde

Sichtbarkeit: Die Sichtbarkeit von Objekten, also Variablen und Funktionen lässt sich anhand folgender Regeln einfach definieren:

1. Ein Objekt kann verwendet werden, wenn es entweder im selben oder bereits in einem der *umgebenen* Blöcke definiert oder es mittels `extern` deklariert (bekannt gemacht) wurde.
2. Ein Objekt eines umgebenen Blocks wird durch ein gleichlautendes Objekt in einem inneren Block überdeckt und damit unsichtbar gemacht.
3. Von einem umgebenen Block aus kann man *nicht* (niemals) auf Objekte innerer Blöcke zugreifen,
4. Alle Objekte einer Datei sind auch außerhalb dieser Datei sichtbar (sie erscheinen in den Exportlisten), es sei denn, sie werden mittels `static` deklariert (dann erscheinen sie nicht in den Exportlisten). Für diese Objekte bildet der Linker den umgebenden Block.

Lebensdauer: Für die noch schnell die beiden folgenden Regeln:

1. Eine Variable lebt solange, wie ihr umgebener Block (die Funktion bzw. das Programm) lebt.
2. Variablen, die innerhalb von Funktionen mittels `static` deklariert werden, überleben den Funktionsaufruf und behalten ihren Wert bei. Diese Variablen werden nicht innerhalb eines Stack Frames auf dem Stack sondern im Data-Segment angelegt. Dadurch werden sie beim Programmstart kreiert und überleben jeden Funktionsaufruf.

56.2 Beispiel für die Sichtbarkeit

Beispiel Datei `file1.c`: Da es nicht einfach ist, sich alles aus ein paar Regeln selbst deduktiv zusammenzusetzen präsentiert dieses Kapitel ein kleines Beispiel, in dem alle Kombinationen enthalten sind.

file1.c

```
1  int x, y = 3;
2  static z = 4;
3
4  static int f1( int i )
5          {
6             int l, z;
7             // Code
8          }
9
10 int vw = 3;      // global mit Initialisierung
11
12 int f2( int y )
13     {
14         static int f1 = 1;
15         // code
16     }
```

Die folgende Tabelle zeigt, auf welche Objekte in den entsprechenden Code-Zeilen zugegriffen werden kann. Um Missverständnisse zu vermeiden, sind die Objekte zusätzlich mit den Zeilennummern versehen, in denen sie definiert bzw. deklariert wurden.

	x.1	y.1	z.2	f1.4	i.4	l.6	z.6	vw.10	f2.12	y.12	f1.14
Zeile 7	ja	ja	nein	ja	ja	ja	ja	nein	nein	nein	nein
Regel	1	1	2	1	1	1	1	1	1	1	1
Zeile 15	ja	nein	ja	nein	nein	nein	nein	ja	ja	ja	ja
Regel	1	2	1	2	3	3	3	1	1	1	1

Beispiel Datei `file2.c`: Das zweite Beispiel sieht wie folgt aus:

file2.c

```
1  extern int x;              // import
2  int f2( int i );           // import
3
4  int f3()
5      {
6          // code
7      }
```

Für diese Datei gilt analog zu oben:

	x.1	f2.2	i.2	f3.4
Zeile 6	ja	ja	nein	ja
Regel	1	1	3	1

Beispiel: Zugriff von `file2.c` auf `file1.c`: Sofern beide Dateien `file1.c` und `file2.c` vom Linker zu einem Programm zusammengefasst werden, kann in Zeile 6 der Datei `file2.c` auf folgende Objekte zugegriffen werden:

Objekte	Zugriff	Kommentar
`x.1`, `y.1`, `vw.10`	ja	Voraussetzung: Import: `extern int x, y, vw;`
`z.2`, `f1.4`	nein	Grund: `static` Deklaration ⇒ kein Export
`i.4`, `l.6`, `z.6`, `y.12`, `f1.14`	nein	Grund: Regel 3

56.3 Beispiel für die Lebensdauer von Variablen

Bei der Beurteilung der Lebensdauer müssen wir schauen, in welchen Segmenten die jeweiligen Variablen angelegt werden. Die generelle Regel ist klar: Variablen auf dem Stack leben so lange wie ihre umgebende Funktion, Variablen im Data oder BSS Segment leben so lange wie das Programm lebt.

Für die Variablen der Datei `file1.c` gilt folgende Segmentzuordnung:

Variablen	Segment	Lebensdauer	Kommentar
`x.1`	BSS	Programm	Globale Variable
`z.2`, `y.1`, `vw.10`	Data	Programm	Globale Variable mit Initialisierung
`i.4`, `l.6`, `z.6`, `y.12`	Stack	Funktion	Formale Parameter, lokale Variable
`f1.14`	Data	Programm	`static`-Variable mit Initialisierung

56.4 Beispiel `static`-Variablen in Funktionen

Als Beispiel sei folgendes Programm gegeben:

```
 1 #include <stdio.h>
 2
 3 int f()
 4     {
 5         static int call = 0;
 6         return ++call;
 7     }
 8
 9 int main( int argc, char **argv )
10     {
11         printf( "return f(): %d\n", f() );
12         printf( "return f(): %d\n", f() );
13         printf( "return f(): %d\n", f() );
14     }
```

Da die Variable `call` mittels `static` definiert wurde, wird sie im Data-Segment angelegt. Da sie bei jedem Funktionsaufruf in Zeile 6 um eins erhöht wird, erscheinen die Zahlen 1, 2 und 3 in der Ausgabe. Würde das Schlüsselwort `static` fehlen, käme immer die Zahl 1 in der Ausgabe an, da die Variable jedesmal wieder auf den Initialwert 0 gesetzt würde. Wer dies nicht glaubt oder sich unsicher ist, führe einfach wieder mal eine Handsimulation durch.

56.5 Verschiedene Anmerkungen

Globale Variablen:

Globale Variablen sind immer ein Thema. Im ersten Moment scheinen sie einiges einfacher zu machen. Beispielsweise kann man von Funktionen aus einfach auf sie zugreifen, ohne sie umständlich übergeben zu müssen. Dies sieht man in C vor allem bei Programmieranfängern, damit sie sich nicht mit der Zeiger-Problematik auseinandersetzen müssen.

Dummerweise ist das Warten globaler Variablen sehr aufwändig. Eine Namens- oder Typänderung kann sehr schnell *sehr* arbeitsaufwändig werden: Wird beispielsweise aus einer globalen Variablen ein Teil eines `struct`s, muss man sehr viel tippen.

Aus obigen Gründen, die aus dem Bereich des Software Engineerings kommen, haben wir bisher in *allen* Beispielen auf globale Variablen verzichtet. Letztlich kann man auf sie verzichten. Will man sie dennoch aus speziellen Gründen verwenden (weil vielleicht der restliche Quelltext einfacher wird), sollte man folgende Designregeln beachten:

1. Globale Variablen mittels `static` vor Zugriffen von außen schützen (Regel 4).
2. Zugriff auf diese Variable mittels selbstgeschriebener Funktionen realisieren.
3. Beispiel:

```
1  static int global_var;
2
3  int get_var()
4      {
5          return global_var;
6      }
7
8  int set_var(int value )
9      {
10         global_var = value;
11         return 1;
12     }
```

Oben angesprochene „Wartungarbeiten“ betreffen dann nur diese wenigen Zeilen.

Lokale Variablen mittels `static`:
Wir erinnern uns, Seiteneffekte in Funktionen sind nicht unbedingt konform mit den Grundgedanken des Software Engineerings. Aber manchmal gibt es doch gute Gründe hierfür. Daher: gut überlegen, ob man sie wirklich benötigt und wenn ja, gut dokumentieren.

Anmerkung zum Schlüsselwort `extern`:
Bei Funktions*deklarationen* kann das Schlüsselwort `extern` angegeben werden oder nicht; es macht kein Unterschied: `int f()` und `extern int f()` sind identisch.

Bei Variablen ist der Unterschied groß: Bei `extern int i` wird *kein* Speicherplatz angelegt (Deklaration) sondern dem Compiler gesagt, dass irgendwo eine Variable `i` existiert. Aber `int i` (ohne `extern`) ist eine Definition, bei der Speicherplatz angelegt wird.

Kapitel 57

`void`: der besondere Datentyp

Zum Schluss dieses Skriptteils widmen wir uns einem ganz speziellen Datentyp, der auch als `void` bekannt ist. Die deutsche Bedeutung des englischen Wortes „void“ ist „unwirksam“, „nichtig“, „ungültig“, „leer“. *„Na, da bin ich ja mal gespannt, ein Datentyp, der ungültig ist. Mittlerweile bin ich ja einiges gewohnt, aber C überrascht mich immer wieder . . .“*

Zugegeben, ein wenig merkwürdig ist die Sache schon, aber mit etwas Struktur versteht man die Idee. Man muss vier Dinge unterscheiden: Variablen vom Typ `void`, Zeiger auf `void`, Funktionen vom Typ `void` und formale Parameterlisten vom Typ `void`. Ursprünglich gab es diesen Datentyp gar nicht. Er wurde vermutlich hinzugenommen, um die Programmiersprache C ein wenig mehr konform in Richtung Software Engineering zu machen. Aber viele kommen zu dem Schluss, dass dieser Versuch gescheitert ist. Dennoch sollten wir uns diesen Datentyp mal eben ansehen.

57.1 Variablen vom Typ `void`

Dieser Punkt ist einfach: Variablen von diesem Typ darf man nicht definieren; derartige Definitionen erzeugen immer einen Übersetzungsfehler.

57.2 Zeiger auf `void`

Ja, Zeiger auf `void` kann man definieren. Dies sind dann Zeiger auf „nichts“ bzw. Zeiger auf „irgendwas.“ Der Unterschied ist folgender: Zeiger auf `void` sind zuweisungskompatibel zu allen anderen Zeigern, alle anderen unterschiedlichen Zeiger sonst nicht. Was das heißt, lässt sich an folgendem Beispiel gut zeigen:

```
1 int  *iptr; char *cptr; void *vptr;
2
3 iptr = cptr;    // warnung da *ip != *cp ist
```

```
4  iptr = vptr;     // ist ok
5  vptr = iptr;     // ist ok
6  cptr = vptr;     // ist ok
7  vptr = cptr;     // ist ok
```

In Zeile 1 werden drei Zeiger unterschiedlichen Typs deklariert. Die Zuweisung in Zeile 3 verursacht eine Warnung, wohingegen die folgenden vier Zuweisungen aus Sicht des C-Compilers OK sind.

„Und wofür braucht man das?“ Eigentlich benötigt man Zeiger auf `void` überhaupt nicht. Zum Tragen kommen sie bei den dynamischen Datenstrukturen, die wir in Skriptteil VII besprechen werden: Hier liefern einige Standardfunktionen Zeiger auf `void` zurück, die man ohne weiteres Zutun jedem anderen Zeiger zuweisen kann.

Ursprünglich haben diese Funktionen Zeiger auf `char` zurückgegeben. Diese Zeiger musste man mittels eines expliziten Cast anderen Zeigern zuweisen; bei Zeiger auf `void` geschieht dies implizit. Und es sei angemerkt, dass ein Cast zwischen zwei Zeigern nichts an der internen Repräsentation ändert, denn eine Adresse ist eine Adresse, nur ist der Compiler dann zufrieden und meckert nicht.

57.3 Funktionen vom Typ `void`

Es kommt vor, dass man tatsächlich Funktionen schreibt, für die man keinen Rückgabewert benötigt. Ein Kandidat hierfür ist die Funktion `printf()`. Hat eine Funktion mit einem „normalen“ Typ keine `return`-Anweisung, gibt der Complier nervtötende Warnungen aus. Entweder ignoriert man diese als Programmierer oder ergänzt ein *unnötiges* `return`, damit der Compiler Ruhe gibt.

Hat man nun eine Funktion vom Typ `void` gibt es eine derartige Warnung nicht. Mit anderen Worten, bei diesen Funktionen kann man auf das `return` verzichten. Dafür gibt es jetzt eine Warnung, wenn man ein `return` mit einem Wert hinschreibt oder gar eine Fehlermeldung, wenn man den Rückgabewert (der ja `void` ist) einer anderen Variablen zuweisen möchte.

57.4 Formale Parameterlisten vom Typ `void`

Bisher haben wir gelernt, dass der Compiler überprüft, ob bei einem Funktionsaufruf die Typen der aktuellen Parameter den Typen der formalen Parametern entsprechen, sofern ihm durch entsprechende Funktionsdeklarationen oder Funktionsdefinitionen die nötigen Informationen vorliegen.

Bei Funktionen *ohne* formale Parameter ist dies nicht so. Hier meckert der Compiler nicht, wenn man trotzdem Parameter übergibt. Durch eine Parameterliste vom Typ `void` erreicht man, dass der Compiler auch hier meckert. Hierzu folgendes Beispiel:

```
1 int f();         // eine Funktion ohne Parameter
2 int g( void );  // eine Funktion mit einer void Parameterliste
3
4 f( 1, 2 );      // akzeptiert der Compiler ohne Warnung!
5 g( 1, 2 );      // hier gibt es eine Fehlermeldung
```

Topic done!

Teil VI

Ein-/Ausgabe

Kapitel 58

Inhalte dieses Skriptteils

Ein- und Ausgabe, ein Mysterium: Alle verwenden sie, und alle wissen ganz genau, wie die einzelnen Anweisungen funktionieren. Als Lehrkörper hingegen müssen wir immer wieder feststellen, dass sich die klaren, individuellen Vorstellungen krass voneinander unterscheiden und häufig nichts mit der Realität zu tun haben.

Bisher haben wir die beiden Funktionen `scanf()` und `printf()` sehr rudimentär verwendet und waren immer froh, wenn es halbwegs funktionierte. Aber genauso werdet ihr hier und da bereits gemerkt haben, dass vor allem die Eingabe nicht so funktioniert, wie Ihr Euch das gedacht habt. Manchmal werdet ihr gedacht haben, dass irgendwelche Eingaben vom PC oder den `scanf()`-Funktionen irgendwie verschluckt wurden. In anderen Fällen, insbesondere bei Programmabbrüchen, werdet ihr das Gefühl gehabt haben, dass irgendwelche Ausgaben irgendwo im Nirwana verschwunden sind. Nun, da der Computer keine Zufallsmaschine ist, sondern deterministisch arbeitet, gibt es in Euren Programmen auch keine Zufälle oder Psiphänomene. Bisher konnten wir *immer* alle Effekte erklären. Das Problem aus Sicht der Lehre ist nun, dass eine vernünftige Erklärung des Ein-/Ausgabesystems, die die wichtigsten Aspekte halbwegs erläutert, nicht nur schwer sondern super schwer ist. Das ist jetzt keine Übertreibung! Selbst sehr weit fortgeschrittene Programmierer haben mit dem Verstehen der Ein-/Ausgabe ihre liebe Mühe.

Aber irgendwie müssen wir Euch zumindest einige wichtige Aspekte näher bringen. Wir haben die Inhalte dieses Skriptteils sehr kontrovers diskutiert und die Kapitel einige Male neu geschrieben. Aber so oder so: die Materie ist einfach *sehr* schwer. Ihr müßt also versuchen, den Text zu lesen, einiges mitzunehmen und vor allem vieles auszuprobieren. Im Zweifel steht der gesamte Lehrkörper bei Fragen zur Verfügung.

Wir fangen damit an, dass wir im nächsten Kapitel einmal einige der weit verbreitesten Mythen aufzählen, die sich um die Ein-/Ausgabe ranken. Diese Mythen haben nichts mit der Wirklichkeit zu tun. Lest sie Euch dennoch durch. Am Ende wisst Ihr wenigstens, was die Ein-/Ausgabe nicht ist. Im Laufe dieses Skriptteils werden wir versuchen, diese Mythen durch (stark vereinfachte) Erläuterungen zurechtzurücken.

Um etwas zurechtrücken zu können, fangen wir in Kapitel 60 damit an, die Konzepte „Datei“ und „Filesystem“ in aller Kürze zu erklären.

Daran anschließend erläutern wir in Kapitel 61 worin sich eigentlich die Schwierigkeiten im Zusammenhang mit Dateien befinden. Aus diesen Schwierigkeiten läßt sich in Kapitel 62 sehr leicht ableiten, was intelligente Geräte-Controller sowie das Betriebssystem machen sollten, damit der Umgang mit Dateien einigermaßen einfach wird.

Obwohl die in Kapitel 62 besprochene Schnittstelle für Dateizugriffe schon recht gut ist, besprechen wir in Kapitel 63, wie die im Standard definierte `FILE` Datenstruktur und das `stdio.h`-Paket aussehen. Die in `stdio.h` definierten Funktionen fassen wir in Kapitel 64 kurz zusammen.

Obwohl das Betriebssystem große Anstrengungen unternimmt, alle Dateien aus Sicht eines C-Programms gleich aussehen zu lassen, verhalten sich dennoch Gerätedateien wie Tastatur und Bildschirm anders als gewöhnliche Dateien. Diesen Aspekt behandeln wir in aller Kürze in Kapitel 65.

Bei allen Erklärungen vereinfachen wir sehr stark. Doch für die wirklich Interessierten stellen wir in Kapitel 66 ein paar weitere, tiefer greifende Details bereit. Doch auch diese Darstellung ist sehr stark komprimiert; man könnte daraus leicht einige 100 Seiten machen. Wer noch mehr wissen will, schaue am besten in die Literatur [1]. Kapitel 67 schließt diesen Skriptteil mit einer kurzen Zusammenfassung ab.

Hier noch einmal die Inhalte dieses Skriptteils im Überblick:

Kapitel	Inhalt
59	Über Mythen und Gruselgeschichten
60	Was ist eine Datei?
61	Herausforderungen bei Dateizugriffen
62	Komplexitätsbewältigung durch Kapselung
63	Die FILE-Schnittstelle
64	Die Standard Ein-/Ausgabe Funktionen
65	Besonderheiten der Terminaleingabe
66	Ein-/Ausgabe aus Sicht des Betriebssystems
67	Dateien: Zusammenfassung

Good luck!

Kapitel 59

Über Mythen und Gruselgeschichten

Um die Ein-/Ausgabe in C ranken sich viele Mythen. Bevor wir in die Inhalte gehen, stellt dieses Kapitel einige dieser Mythen zusammen. Diese zeigen deutlich, dass das Unverständnis über die Ein-/Ausgabe in C wesentlich größer als das Verständnis ist. Wir wollen hier gleich noch betonen, dass wir uns die folgenden Mythen nicht ausgedacht haben, sondern sie regelmäßig in den Übungen und Klausuren sehen.

1. Wenn man Zahlen und Zeichen mittels `scanf()` einlesen möchte, werden manchmal verschiedene Eingaben verschluckt. Auch das Zwischenspeichern hilft da nichts. Also, Letzteres ist tatsächlich wahr, aber der erste Teil der Aussage ist grundsätzlich falsch. Die Funktion `scanf()` hat noch nie eine Eingabe verschluckt.

2. Ein Zeiger `p` vom Typ `FILE *` zeigt offensichtlich auf ein Array. Dann kann ich das n-te Zeichen dieser Datei mittels `p[ n ]` einfach lesen oder auch verändern. Der erste Satz stimmt, doch der Rest ist kompletter Nonsens.

3. *„`gets()` und `fgets()` sind die gleichen Funktionen. Ich benutze `gets()`, weil sie weniger Schreibarbeit erfordert.“* Wer programmiert und `gets()` benutzt, sollte sein Abitur zurückgeben und niemals einen Universitätsabschluss erhalten.

4. Ein Zeiger `p` vom Typ `FILE *` verweist auf die geöffnete Datei, insbesondere auf die aktuelle Position innerhalb der Datei. Mittels `p++` bzw. `p--` kann man in der Datei nach vorne und hinten gehen. Das ist leider völlig falsch und bedarf einiger etwas längerer Erläuterungen.

5. Da ein Zeiger `p` vom Typ `FILE *` auf eine geöffnete Datei zeigt, kann man mittels `sizeof( p )` die Größe der offenen Datei bestimmen. Das ist so falsch, dass wir als Lehrkörper sprachlos sind. Zur Erinnerung: Die Compiler-Funktion `sizeof()` wird beim Übersetzen ausgewertet, bestimmt die Zahl der Bytes, die ein derartiger Zeiger im Arbeitsspeicher belegt, und hat *nichts* mit Dateien zu tun.

6. File-Pointer und File-Nummern sind eigentlich das Selbe. File-Nummern sind die

Indizes in einer ominösen File-Tabelle, File-Pointer sind die Adressen dieser Elemente. Nein, leider komplett falsch. File-Nummern sind etwas ganz anderes und Teil des Betriebssystems.

Obige Mythen gehören alle in dem Bereich der Fabeln und Märchen und haben nichts mit der Realität zu tun. Wir werden sie erst mal so stehen lassen und hoffen, dass wir sie in diesem Skriptteil entmystifizieren können.

Kapitel 60

Was ist eine Datei?

Normale Dateien haben wir schon zu genüge kennengelernt. Alle unsere Programme, egal ob Quelltext oder ausführbarer Maschinencode, waren in Dateien abgespeichert. Auf unseren Rechnern gibt es aber noch eine zweite Form von Dateien. In modernen Betriebssystemen werden auch Geräte, wie beispielsweise Tastatur und Bildschirm als Dateien behandelt: In diesem Kapitel erklären wir grob, wie diese Dateien aufgebaut sind und auf ihren Datenträgern organisiert werden.

60.1 Normale Dateien

Warum eigentlich reguläre Dateien? Diese Frage klingt vielleicht naiv, aber sie ist dennoch lohnenswert. Also, warum werden Dateien auf externen Datenträgern benötigt? Hierfür gibt es mindestens zwei Gründe. Erstens hat man viel mehr Daten als Arbeitsspeicher, sodass diese Daten nicht alle in den Arbeitsspeicher passen. Egal wie groß die externen Platten auch sein mögen, so sind sie notorisch voll. Der zweite Grund ist das Ausschalten des PCs. Bei (fast) allen halbleiterbasierenden PCs ist der Arbeitsspeicher flüchtig, sodass die Daten weg sind, wenn man den PC ausschaltet. Entsprechend muss man vor dem Ausschalten seine Daten in einer Datei auf einer Platte retten.

Was steht in einer Datei? Diese Frage ist ganz einfach zu beantworten: Bei fast allen modernen Betriebssystemen wie Windows und Linux werden die Daten eins-zu-eins aus dem Arbeitsspeicher in eine externe Datei geschrieben. D.h., dass sowohl im Arbeitsspeicher als auch in externen Dateien acht Nullen/Einsen zu einem Byte zusammengefasst werden. Insofern kann man jedes einzelne Byte einer Datei als ein `char` auffassen, was aber nicht heißt, dass die Daten einer Datei auch als Text lesbar sind. Es kann gut sein, wie es beispielsweise bei Maschinencode ist, dass die einzelnen `char`s völlig unverständlich sind.

Was für Datenträger werden für Dateien verwendet? Und schon wird's schwierig. Werfen wir einen Blick zurück und fangen an mit „Es war einmal. . .“

Lochstreifen: Die ersten Computer hatten so Lochstreifen. Jedes Zeichen etc. wurde mittels fünf bis acht Löchern bzw. „Nichtlöchern“ kodiert. Diese Lochstreifen und deren Löcher waren so groß, dass man jedes einzelne Byte gut mit bloßem Auge sehen konnte. Der Operateur hat den Lochstreifen eingelegt und schon ging es los. Aber Folgendes sollte klar sein: Ein derartiges Medium ist mit einer Lesegeschwindigkeit von unter 10 Bytes pro Sekunde und einer Kapazität von 250 Bytes pro Meter Lochstreifen nach unserem Verständnis für heutige Anwendungen nicht geeignet.

Lochkarten: Etwas besser waren Lochkarten. Bis zu 80 Zeichen haben auf einer Lochkarte Platz. Eine Datei war dann ein richtig schöner Stapel. Bezüglich Lesegeschwindigkeit (etwa 10-35 Karten pro Sekunde) und Speicherkapazität sind Lochkarten wesentlich besser als Lochstreifen, aber für unsere Verhältnisse völlig unbedeutend. Interessant ist vielleicht: bei einer ungefähren Dicke von 0,2 mm, konnte man pro Meter Lochkartenstapel ungefähr 400 KB speichern. Wie viele Meter bräuchte man für eine DVD?

Magnetbänder: Später gab es Magnetbänder, wie man sie beispielsweise aus den Science Fiction Filmen der 70er Jahre kennt. Diese konnten wesentlich mehr speichern und waren auch schneller. Aber für unsere Bedürfnisse immer noch sehr, sehr langsam. Dennoch, moderne Magnetbänder werden immer noch für Backups verwendet da sie vergleichsweise preiswert und robust sind.

Magnetplatten: In den 80ern fanden die Magnetplatten eine immer größer werdende Verbreitung. Mitte der 80er hat man sich als Arbeitsgruppe über 100 MB gefreut. Diese 100 MB waren aber für alle 10 Mitarbeiter ;-) Das Prinzip der Magnetplatten hat sich bis heute erhalten. Es handelt sich um mehrere Platten, die sich mit ca. 10.000 Umdrehungen pro Minute drehen. Nach dem ein Schreib-/Lesekopf in die richtige Spur gebracht wurde, kann der Datentransport durchgeführt werden. Heutzutage haben Plattenlaufwerke eine Kapazität von bis zu einigen Tera Bytes. Bei diesen Plattenlaufwerken werden alle Platten und Schreib-/Leseköpfe synchron bewegt.

Seit einigen Jahren werden in teuren PCs und Laptops die Plattenlaufwerke mehr und mehr durch SSDs (Solid State Disks) abgelöst. Dabei handelt es sich um Halbleiterspeicher wie wir sie aus mp3-Spielern kennen. Von außen betrachtet, verhalten sich SSDs wie herkömliche Plattenlaufwerke.

Wie werden Dateien organisiert? Die Beantwortung dieser Frage hängt in erster Linie vom verwendeten Datenträger ab.

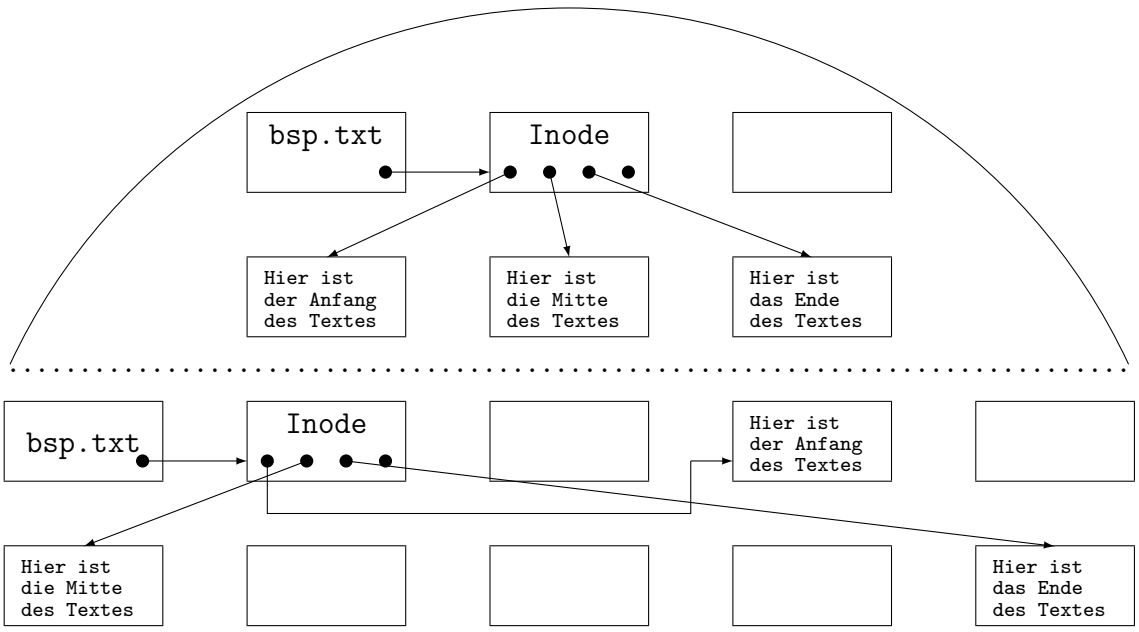
Lochstreifen und Magnetbänder: Bei den Lochstreifen und Magnetbändern wurden die Daten einfach Byte für Byte auf den Datenträger geschrieben. Ab und zu gab es eine kleine Lücke und am Ende eine Datei-Endekennung, was wir heutzutage als `End-of-File` (`EOF`) kennen. Wollte man eine Datei wieder einlesen, musste man nur den Namen des Bandes kennen und wissen, die wievielte Datei man auf diesem Band lesen wollte. War ja bei ca. 10 Dateien pro Band auch gar kein Problem.

Magnetplatten: Bei Magnetplatten ist die Sachlage völlig anders. Das Datenvolumen ist

sehr groß und die Platten drehen sich zumindest im Vergleich zu einer Musikkassete sehr schnell unter dem Schreib-/Lesekopf hindurch. Um hier die einzelnen Bytes wiederzufinden und eine halbwegs erträgliche Transferrate zu erzielen werden die einzelnen Bytes zu Blöcken konstanter Größe zusammengefasst. Die Größe eines Datenblocks liegt heutzutage bei ungefähr 1 KB bis 4 KB; in konkreten Konfigurationen kann die tatsächliche Blockgröße davon natürlich abweichen. Diese Datenblöcke werden über die gesamte Platte verteilt. Um die Daten wiederzufinden werden üblicherweise die ersten Blöcke einer Platte als (Platten-) Verzeichnis verwendet. Über dieses Plattenverzeichnis findet man alle Dateien sowie die zugehörigen Datenblöcke. Für die Interessierten: Die Datenstruktur, die eine einzelne Datei beschreibt, nennt man *Inode*.

Die uns bekannten Verzeichnisse/Ordner/Directories sind wie die eigentlichen Dateien Bestandteile dieser Plattenverzeichnisse. Die Gesamtstruktur aus Plattenverzeichnis und Datenblöcken nennt man *File System* (Filesystem). Bekannte Vertreter von Filesystemen sind Ext2, Ext3 und NTFS. Je nach Filesystem sind die Verzeichnisse und Beschreibungen der einzelnen Dateien unterschiedlich aufgebaut. Will man einen Datenträger ansprechen, muss der PC und das Betriebssystem mit diesem Typ von Filesystem auch umgehen können. Daher kann man beispielsweise Linux Filesysteme nicht unter Windows benutzen. Auf heutigen Plattenlaufwerken sind häufig einige Tausend Dateien in verschiedenen Verzeichnissen und Unterverzeichnissen organisiert. Unten stehende Grafik zeigt beispielhaft die Organisation einer Datei `bsp.txt`, die aus drei Datenblöcken besteht. Der obere Teil deutet die Verteilung auf der Platte an, wie wir sie empfinden. Der untere Teil hingegen illustriert, wie die einzelnen Blöcke in Wirklichkeit verteilt sein könnten. Auf dem Laufwerk hat jeder einzelne Block eine Nummer, was den Indizes eines Arrays sehr ähnelt.

Organisation einer Datei auf einem Plattenlaufwerk



60.2 Gerätedateien

Wie eingangs erwähnt verstehen wir hierunter Geräte wie Tastatur und Bildschirm. Der Bildschirm ist dabei relativ unproblematisch. Aus einem C-Programm heraus können wir einfach auf den Bildschirm ausgeben und alles ist gut.

Die besondere Herausforderung liegt in der Gerätedatei namens Tastatur. Warum ist das so? Nun, davor sitzt ein Nutzer. Und dieser Nutzer ist inhärent unberechenbar. Er macht was er will und wann er will. Etwas überspitzt gesagt: Aus Sicht eines Programmierers ist so ein Nutzer ein Super-GAU, denn alles was er falsch machen kann, macht er auch falsch. Und wir als Programmierer müssen damit irgendwie umgehen.

Eine weitere Problematik folgt unmittelbar aus dem Nutzer, der vor dem Gerät sitzt und etwas eingeben soll. Uns als Programmierer/Programm bleibt nichts anderes übrig, als auf die Nutzereingaben zu warten. Da wir weder den Nutzer kennen noch über übersinnliche Fähigkeiten verfügen wissen wir nicht, wann wie viel Zeichen eingegeben werden, was wiederum bedeutet, dass so eine Gerätedatei gar keine Größe hat; wie oben besprochen hat im Gegensatz dazu eine reguläre Datei wie beispielsweise `bsp.txt` immer eine Größe denn nach dem Abspeichern besteht sie nun mal aus x Bytes.

Ferner müssen wir berücksichtigen, dass der Nutzer Fehler macht. Er tippt etwas ein, löscht es wieder mit der Backspace Taste, tippt erneut etc. Auf all diese Aktionen müssen wir (unser C-Programm oder auch das Betriebssystem) in angemessener Form reagieren; ein Computerabsturz nach jedem Tippfehler würde nicht direkt auf Gegenliebe stoßen.

60.3 Zusammenfassung

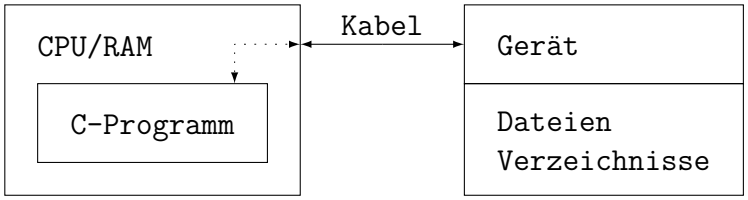
Wir müssen zwei Sorten von Dateien auseinanderhalten: normale (reguläre) Dateien und Gerätedateien. Als kleinste Einheit haben alle Dateien ebenso wie der Arbeitsspeicher ein Byte. Reguläre Dateien werden auf Plattenlaufwerken, die wir im Rahmen unserer Lehrveranstaltung berücksichtigen, in Form von Blöcken zu 1 KB oder einem Vielfachen davon zusammengefasst. Um die Dateien und vor allem auch deren Daten wiederzufinden, wird auf einem Plattenlaufwerk ein Filesystem installiert. Ein derartiges Filesystem besteht aus einem ersten Bereich, das man (Platten-) Verzeichnis nennt, und einem zweiten Bereich, in dem sich die eigentlichen Datenblöcke befinden. Im Verzeichnisbereich befinden sich alle Angaben zu den Dateien und allen Unterverzeichnissen/Ordner/Directories. Über diese Verzeichnisstruktur kann man jederzeit auf die einzelnen Datenblöcke zugreifen. Gerätedateien hingegen besitzen nicht unbedingt eine Größe und erfordern zusätzliche Überprüfungen der Daten durch ein C-Programm oder das Betriebssystem.

Kapitel 61

Herausforderungen bei Dateizugriffen

Im vorherigen Kapitel haben wir gesehen, was Dateien sind und wie sie auf externen Datenträgern organisiert werden. Nun stellt sich die Frage, was wir alles machen *müssten*, um an die Daten heranzukommen. Wir betrachten dabei den *hypothetischen* Fall, dass wir die Datenträger ohne Hilfe des Betriebssystems ansprechen. Daraus resultieren verschiedene Herausforderungen, aus denen wir recht einfach ableiten können, warum wir ein Betriebssystem benötigen und es zwischen die Geräte und Nutzerprogramme schalten. Die Sichtweise, die wir in diesem Kapitel einnehmen, ist also in etwa wie folgt:

Kommunikation zwischen CPU und Datenträgern



61.1 Aufgaben und Herausforderungen

Bei Zugriffen auf Dateien müssen folgende Herausfoderungen bewältigt werden:

Ansprechen der Geräte:
Wir müssten wissen, wie wir von unserem C-Programm aus die einzelnen Geräte ansprechen müssen, welche Informationen wir übermitteln müssen und wie wir die Daten aus den ankommenden Datenblöcken herausfiltern können. Vor allem müssten wir jedesmal wissen, was für ein Gerät gerade angeschlossen ist und wie das „Protokoll“ im Einzelfalle aussieht.

Auffinden der Datenblöcke:
Um den richtigen Datenblock zu finden, bräuchten wir genaue Kenntnis über das

aktuelle Filesystem. Nur so können wir die Verzeichnisstruktur richtig interpretieren und die richtigen Datenblöcke auswählen.

„Bearbeiten“ des Datenblocks:
Es könnte unter Umständen notwendig sein, die gelesenen bzw. zu schreibenden Datenblöcke nachzubearbeiten. Ein derartiges Bearbeiten könnte beispielsweise darin bestehen, zusätzliche Information zur Datensicherheit einzuarbeiten.

Sicherheitsmechanismen:
Beim Verständnis der zu bewältigenden Herausforderungen sollten wir auch daran denken, dass ein PC prinzipiell von mehreren Personen verwendet werden kann. Dies bedeutet eigentlich, dass wir gewisse Sicherheitsmechanismen benötigen, da eine Person nicht die Daten einer anderen Person lesen und/oder verändern können sollte. Ohne zusätzliche Unterstützung können wir das einfach nicht (selbst) erreichen.

Datenintegrität:
Ein weiteres Problem kommt auf, wenn wir das Plattenverzeichnis direkt lesen oder beschreiben könnten. Hier könnte ein kleiner Programmierfehler dazu führen, dass die Integrität des Plattenverzeichnisses zerstört wird, sodass *keine* Daten mehr auffindbar sind. Schon alleine aus praktischen Gründen wäre hier kompetente Unterstützung notwendig.

Geschwindigkeit:
Viele gehen regelmäßig in einen der Elektronikmärkte, um sich nach neuen, möglichst schnellen Rechnern umzuschauen. Da stellt sich die Frage, wie schnell eigentlich ein Plattenlaufwerk ist. Hierzu folgende Überlegung: Um einen Datenblock zu lesen oder zu beschreiben, muss der Schreib-/Lesekopf an die richtige Stelle bewegt werden. Das kann selbst bei kleinen Platten mit bis zu 2 cm Durchmesser etwa 20 ms dauern. Anschließend müssen wir den richtigen Zeitpunkt abwarten, damit wir den richtigen Datenblock aus der Masse der am Schreib-/Lesekopf vorbeifliegenden Blöcke herausnehmen können. Bei 10.000 Umdrehungen pro Minute müssten wir durchschnittlich 3 ms warten. Diese Zeit ist im Vergleich zur Zeit, die wir für das Positionieren des Schreib-/Lesekopfes benötigen, recht klein, sodass wir sie vernachlässigen können.

Aus den Zeiten, die sich durch rein mechanische Bewegungen ergeben, können wir höchstens bis zu 50 Datenblöcke pro Sekunde lesen, sofern sie über das Plattenlaufwerk verteilt sind! Bei einer Blockgröße von 1 KB macht das gerade mal 50 KB pro Sekunde. Wie schnell ist dies im Vergleich zur Verarbeitungsgeschwindigkeit der CPU, wenn diese mit 1 GHz getaktet wird? Die Antwort lautet: Das Lesen (bzw. Schreiben) eines einzigen Bytes dauert etwa 20.000 mal länger als die Ausführung einer einfachen Anweisung durch die CPU. Für den Fall, dass sich alle Datenblöcke auf der selben Spur befinden, könnten wir bis zu etwa 333 unterschiedliche Datenblöcke pro Sekunde lesen.

61.2 Maßnahmen zur Verbesserung

Natürlich sind wir nicht die ersten, denen obige Probleme bewusst sind. Daher wurden in den letzten Jahren viele Verbesserungen entwickelt. Diese Maßnahmen betreffen in erster Linie die Geräte selbst und die Realisierung einiger Funktionen durch ein dazwischengeschaltetes Betriebssystem:

Intelligente Controller:
Um die CPU und alle Programme zu entlasten, sind in den meisten Geräten Mikrocontroller eingebaut, die das gesamte „Handling“ des Datenträgers übernehmen. Die CPU muss dem Gerät nur mitteilen, welchen Datenblock sie lesen möchte. Den Rest erledigt das Gerät völlig eigenständig, sodass sich die CPU mit anderen Dingen beschäftigen kann. D.h., dass viele Spezifika des Datenträgers durch einen intelligenten Geräte-Controller gekapselt sind und dass man auf einfache Weise ganze Datenblöcke lesen und schreiben kann.

Betriebssystem und Datensicherheit:
Durch die Installation eines Betriebssystem wird verhindert, dass irgendein (beliebiges) Nutzerprogramm direkt auf die Datenblöcke zugreifen kann. Das Betriebssystem weist jeder Datei einen Eigentümer zu und überprüft bei jedem Datenzugriff, ob das Programm überhaupt die Berechtigung zum Zugriff auf die Daten hat. In den meisten heutigen Betriebssystemen ist es so, dass auch zwischen Leseberechtigung und Schreibberechtigung unterschieden wird. Mit anderen Worten: Es kann sein, dass ein Nichteigentümer eine Datei möglicherweise lesen aber nicht beschreiben darf.

Betriebssystem und Filesystem:
Für das Betriebssystem ist es relativ einfach, die Spezifika der unterschiedlichen Filesysteme zu berücksichtigen. Natürlich muss alles implementiert werden. Aber nur einmal für alle Programme. Wenn das geschafft ist (und das ist es in der Regel), braucht sich kein Nutzerprogramm mehr darum kümmern.

Betriebssystem und Datenintegrität:
Durch das Verhindern direkter Zugriffe auf die Datenblöcke kann das Betriebssystem auch die Datenintegrität zusichern: Ein Nutzerprogramm kann seine Daten dem Betriebssystem übergeben. Das Betriebssystem wird daraufhin die richtigen Blöcke lesen und/oder beschreiben, sodass die Integrität des externen Datenträgers (aus unserer C-Programmierersicht) automatisch gewährleistet ist.

Geschwindigkeit durch Geräte-Controller und Betriebssystem:
Sowohl Gerätehersteller als auch Betriebssystementwickler investieren sehr viel Arbeit in das möglichst schnelle Ansprechen der Geräte, denn dies ist immer noch ein recht wichtiges Verkaufsargument. Heutige Geräte-Controller sind häufig so intelligent, dass sie beim Lesen nicht nur einen Datenblock lesen sondern alle Datenblöcke einer kompletten Umdrehung (was man Spur nennt). Alle diese Datenblöcke werden in einen internen Cache geschrieben, sodass der nächste Datenblock gar nicht mehr

von der Platte gelesen werden muss, sondern (hoffentlich möglichst oft) direkt aus dem Cache geholt werden kann.

Das gleiche macht das Betriebssystem. In der Regel hat es mehrere Laufwerke, mehrere Prozesse, mehrere Nutzer etc. Daher verwalten heutige Betriebssysteme einen eigenen Cache, um nach Möglichkeit gar nicht erst in die Verlegenheit zu kommen, ein externes Gerät ansprechen zu müssen. Ferner hilft dieser Cache enorm bei der Gewährleistung der Datenintegrität.

Bei all diesen Verbesserungen haben wir leider immer noch nicht das Problem gelöst, dass Dateien deutlich größer als der verfügbare Arbeitsspeicher sein können. Dazu kommen wir im nächsten Kapitel.

Kapitel 62

Komplexitätsbewältigung durch Kapselung

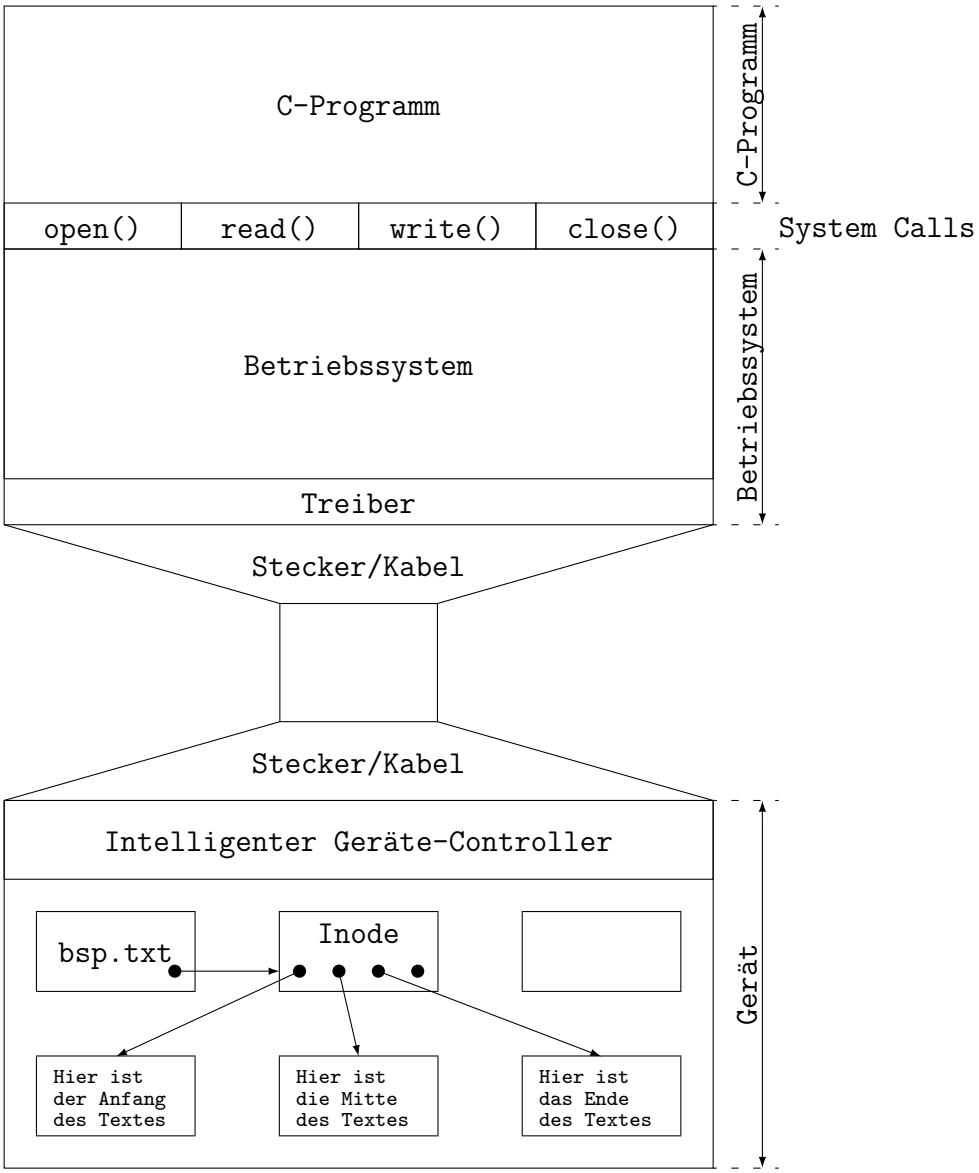
Im vorherigen Kapitel haben wir darüber geredet, wo die Herausforderungen im Zusammenhang mit Dateien liegen und was Gerätehersteller und Betriebssystementwickler so alles gemacht haben, um diese Probleme zu beheben. Vor allem sollte durch die Diskussion klar geworden sein, dass es für uns als C-Programmierer vollkommen sinnlos wäre, direkt auf die externen Geräte (Datenträger) zuzugreifen: der Zugriff wäre bei weitem viel zu komplex und bei weitem viel zu ineffizient. In diesem Kapitel besprechen wir, wie beide Probleme durch ein gut gestaltetes Betriebssystem weitestgehend gelöst werden können.

62.1 Gerät ⇔ Betriebssystem ⇔ Nutzerprogramm

Kapselung ist eine typische ingenieursmäßige Methodik. Die Idee dabei ist, dass man eine (Abstraktions-) Schicht einfügt, die nach außen eine leistungsfähige Schnittstelle zur Verfügung stellt, in dem sie viele Details verbirgt und einzelne Funktionalitäten zu neuen leistungsfähigen Basisoperationen zusammensetzt. Für den Zugriff auf Dateien ist es üblich, zumindest zwei Abstraktionsschichten einzufügen, indem die Geräte einen intelligenten Controller erhalten, und ein Betriebssystem, das die Geräte von unseren C-Programmen trennt. Im Wesentlichen sieht dies wie folgt aus (siehe auch die folgende Abbildung):

Geräte:

Geräte, die für uns einen externen Datenträger bereitstellen, erhalten in der Regel einen intelligenten Controller, der meist durch einen Mikrocontroller realisiert wird. Dieser Controller übernimmt alle physikalisch/elektrischen Details wie Spannungsversorgung, Motorsteuerung, Timing, Signalaufnahme und -analyse etc. Diese Schnittstelle stellt meistens folgende Operationen nach außen zur Verfügung: Starten des Gerätes, Lesen eines angegeben Blocks, Schreiben eines angegebenen Blocks und Anhalten des Gerätes. Diese vier Operationen reichen für die nächste Schicht völlig



aus, können aber durch weitere Operationen angereichert werden.

Betriebssystem:

Das Betriebssystem kümmert sich um alle Details, die mit dem Filesystem zu tun haben. Das bedeutet, das Betriebssystem liest die einzelnen Verzeichnisse der Datenträger, liest die Beschreibungen der einzelnen Dateien (die Inodes) und ermittelt, in welchen Blöcken sich die eigentlichen Daten befinden. Ferner aktualisiert das Betriebssystem bei jedem Dateizugriff noch einige statistische Parameter, die ebenfalls in der Inode abgelegt werden. Das Betriebssystem führt alle diese Operationen in einer Art und Weise durch, das gleichzeitig die Datenintigrität sowie die Berücksichtigung aller Zugriffsrechte gewährleistet ist. Ferner ist das Betriebssystem so ausgelegt,

dass es alle seine internen Datenstrukturen gegen direkte Zugriffe von außen schützt. Die Funktionen der Schnittstelle zwischen Betriebssystem und Nutzerprogramm werden auch System Calls genannt und stellen spezifische, gut geschützte Operationen nach außen zur Verfügung. Für Dateizugriffe werden in der Regel mindestens folgende Operationen angeboten: Öffnen einer Datei (`open()`), Lesen von Daten (`read()`), Schreiben von Daten (`write()`) und Schließen einer Datei (`close()`). Diese einfache Schnittstelle reicht für die meisten Anwendungsfälle völlig aus. Aus Effizienzgründen wird in der Regel diese Schnittstelle durch weitere Operationen angereichert.

Um es vielleicht noch einmal zu wiederholen: Die Schnittstelle, die das Betriebssystem für Dateizugriffe zur Verfügung stellt, ist so ausgelegt, dass sie für möglichst viele Anwendungen geeignet ist. Mit anderen Worten: Die Schnittstelle soll möglichst alles ermöglichen und zwar sauber, sicher und effizient. Dabei ist zu berücksichtigen, dass der letzte Aspekt nur im Rahmen von Sicherheit und Integrität realisiert werden kann. Ferner sollte jedem bewusst werden, dass im Sinne einer möglichst großen Flexibilität die Operationen dieser Schnittstelle relativ primitiv sind. Beispielsweise können nur Zeichen ausgegeben werden; die Umwandlung von `int`-Werten in entsprechende ASCII Zeichen muss an einem anderen Ort geschehen. Letztlich ist die Datei-Schnittstelle so ausgelegt, dass sie von möglichst allen Programmiersprachen verwendet werden kann.

62.2 Die abstrakte Sicht auf eine Datei

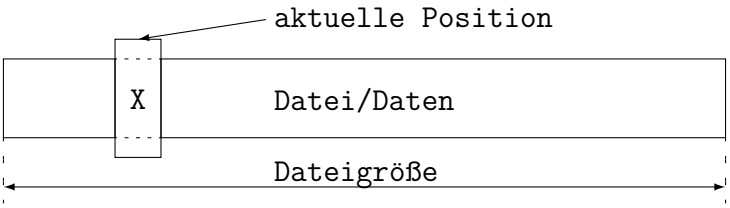
Wie oben erklärt wird durch das Betriebssystem eine neue Abstraktionsschicht eingefügt, die auch die Sichtweise auf eine Datei ändert. Ist eine Datei erst einmal geöffnet, ähnelt sie von einem Nutzerprogramm aus gesehen sehr stark einem großen `char`-Array. Doch gibt es einen kleinen, aber dennoch *wesentlichen* Unterschied:

Die gesamte Datei befindet sich eben nicht im Arbeitsspeicher (wie sonst alle anderen Variablen) sondern auf einem externen Datenträger. Dadurch sind die Zugriffe auf die einzelnen Stellen immer noch recht langsam, da die Daten erst eingelesen oder geschrieben werden müssen. Oder auch anders ausgedrückt: Die Inhalte dieses externen Speichermediums können nur stückweise eingelesen bzw. ausgegeben werden.

Um eine Datei, die auf einem externen Datenträger liegt, stückweise einlesen (bzw. ausgeben) zu können, gibt es prinzipiell zwei unterschiedliche Ansätze: Entweder gibt man bei *jeder* Lese- oder Schreiboperation die Stelle an, von der gelesen bzw. geschrieben werden soll, oder man bezieht alle Lese- und Schreiboperationen auf eine aktuelle Position, die nach jeder Operation vom Betriebssystem automatisch aktualisiert wird. Linux und Windows haben sich für die zweite Variante entschieden. Die folgende Grafik versucht dies ein wenig zu visualisieren: ein Schreib-/Lesefenster gleitet über die Datei und liest bzw. schreibt immer dasjenige Zeichen, das sich gerade unter dem Schieber befindet. Nach dem Lesen bzw. Schreiben *jedes* einzelnen Zeichens wird dieser Schieber vom Betriebssystem

automatisch um eine Stelle nach rechts verschoben.

Lesen und Schreiben ab einer aktuellen Position



Damit alles in unserem Sinne funktioniert, verwaltet das Betriebssystem für jede unserer geöffneten Dateien eine Nummer, die Größe der Datei und eine aktuelle Lese-/Schreibposition, auf die sich alle Ein-/Ausgabeoperationen beziehen. Die Nummer der geöffneten Datei wird auch File Descriptor (FD) genannt.

62.3 Verwendung der Dateischnittstelle

Beispielhafte Verwendung: Wie gesagt, die Verwendung obiger Dateischnittstelle, die vom Betriebssystem bereitgestellt wird, ist recht einfach ausgelegt und kann von jedem C-Programm verwendet werden. Bevor wir lange darüber reden, hier lieber ein einfaches Beispiel, das wir anschließend wie üblich im Detail besprechen.

```
 1 #define MSG1        "hello, world\n"
 2 #define MSG2        "fatal error\n"
 3
 4 int main( int argc, char **argv )
 5     {
 6         int i, fd;
 7         char buf[ 20 ];
 8         i = read( 0, buf, sizeof( buf ) ); // read from "stdin"
 9         write( 1, MSG1, sizeof( MSG1 ) );  // write to "stdout"
10         write( 2, MSG2, sizeof( MSG2 ) );  // write to "stderr"
11         fd = open( "test.txt", 0101, 0644 );  // open for write
12         if ( fd != -1 )
13             write( fd, ":-)\n", sizeof( ":-)\n" ) );
14     }
```

Zeile 8: Hier werden maximal 20 Zeichen von der Tastatur gelesen. Der `read()`-System Call gibt die Anzahl der gelesenen Zeichen zurück.

Zeile 9: Schreiben einer Nachricht auf den Bildschirm (Standardausgabe).

Zeile 10: Schreiben einer Nachricht auf den Bildschirm (Standardfehler).

Zeile 11: Öffnen der Datei `test.txt` zum Schreiben. Den Rückgabewert vom Typ `int` nennt man *File Descriptor (FD).* Die beiden Konstanten `0101` und `0644` muss man in der Dokumentation nachlesen.

Zeile 13: Schreiben eines Smilys in die gerade geöffnete Datei.

Zeilen 8, 9, 10, 11, 13: Den jeweils ersten Parameter dieser Zugriffsfunktionen nennt man File Descriptor (FD). Die File Descriptoren `0`, `1` und `2` sind für die Standardeingabe, Standardausgabe und Standardfehler vorinitialisiert.

Die Standardein-/ausgabe: Die Vorinitialisierung der File Descriptoren `0`, `1` und `2` wird bereits vor Programmstart automatisch vom aufrufenden Programm initiiert und vom Betriebssystem automatisch durchgeführt. Durch diesen Automatismus kann man direkt von der Tastatatur lesen und auf den Bildschirm schreiben; natürlich kann man diese Descriptoren in Dateien umlenken, wie wir es bereits in vielen Übungsaufgaben gemacht haben.

Beurteilung der Schnittstelle: Wie oben beschrieben, ist die vom Betriebssystem angebotene Dateischnittstelle universell gestaltet und damit recht einfach gehalten. Obwohl man sie recht einfach verwenden kann, hat sie zwei entscheidende Nachteile:

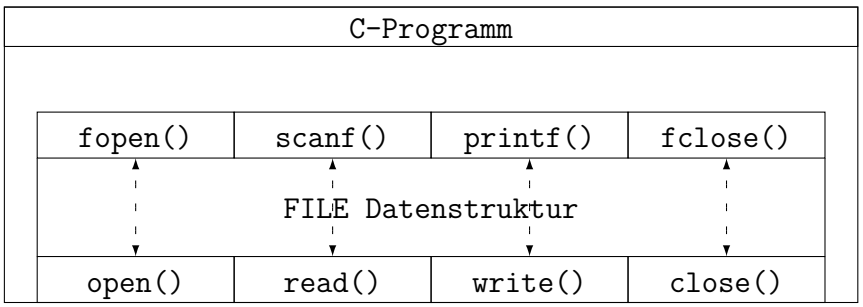
1. Die Schnittstelle ist einfach zu primitiv. Wir können nicht einmal einfache Zahlen lesbar darstellen. Ebenso können wir nur zeichenweise einlesen, was auch nicht besonders leistungsfähig ist.
2. Der Aufruf eines System Calls wie `read()` oder `write()` ist recht teuer im Sinne von benötigter Rechenzeit.

Aufgrund dieser beiden gravierenden Nachteile wurde im C-Standard eine weitere Abstraktionsschicht definiert, die wir im nächsten Abschnitt kurz vorstellen. Durch diese als `FILE`-bekannte Abstraktionsschicht brauchen wir die vom Betriebssystem angebotene Schnittstelle in der Regel nicht zu verwenden.

Kapitel 63

Die `FILE`-Schnittstelle

Im vorherigen Kapitel haben wir die Dateischnittstelle diskutiert, wie sie vom Betriebssystem bereitgestellt wird. Obwohl diese Schnittstelle für alle Anwendungen ausreicht, hat sie dennoch zwei Schwachstellen: sie ist nicht besonders komfortabel und für das Nutzungsverhalten vieler C-Programme recht langsam. Aus diesem Grund wurde von Beginn an eine weitere Abstraktionsschicht realisiert, die unter dem Begriff `stdio.h` allgemein bekannt ist. Diese `.h`-Datei enthält eine Datenstruktur, die als `FILE` bekannt ist, sowie eine Reihe recht leistungsfähiger Ein-/Ausgabefunktionen. Die Gesamtstruktur sieht wie folgt aus:



Das Gesamtkonzept: Das Gesamtkonzept besteht aus drei Komponenten, „unten“ den System Calls, darüber einer Array-Datenstruktur, bei dem die Elemente `structs` vom Typ `FILE` sind, und darüber einer Reihe neuer Ein-/Ausgabefunktionen, die fast alle ein „f“ am Ende ihres Namens haben. Die `FILE` Datenstruktur dient als Mittler zwischen den neuen Funktionen und den System Calls. Diese Konstruktion mach die Ein-/Ausgabe wesentlich leistungsfähiger, da diverse Möglichkeiten zum Konvertieren von Datentypen angeboten werden. In den Übungen haben wir beispielsweise den Wert einer `int`-Variablen `i` mittels `printf( "%d\n", i )` ausgegeben. Und wir wissen auch, wie wir Werte von den Typen `double` und `char` sowie Zeichenketten und Zeigerwerte ausgeben können.

Funktionsweise: Jede geöffnete Datei hat in der `FILE`-Datenstruktur ein eigenes Element vom Typ `FILE`. Dieser Datentyp sieht *konzeptuell* wie folgt aus (die tatsächliche Implementierung ist Sache des Betriebssystems):

```
1 // Eine *konzeptuelle* Sicht auf die Struktur FILE
2 typedef struct {
3                 int fd;             // the Unix file descriptor
4                 int bsize;     // the number of bytes in buffer
5                 int nextc;               // next char in buffer
6                 char buf[ BS ];     // might be somewhere else
7         } FILE;
```

Kern dieses `structs` ist ein Puffer, den wir oben beispielhaft `buf` genannt haben. Die Arbeitsweise ist nun so, dass alle Ein-/Ausgabefunktionen des Pakets `stdio.h` nicht direkt die `read()` und `write()` System Calls aufrufen, sondern aus diesen Puffern lesen bzw. in diese Puffer schreiben. Erst wenn diese Puffer beim Lesen leer oder beim Schreiben voll werden, werden die entsprechenden System Calls automatisch aufgerufen. Oder anders gesagt: um dieses Detail braucht man sich nicht zu kümmern. Durch das Zwischenspeichern der Daten in diesen Puffern wird die Zahl der System Call Aufrufe drastisch verringert und der Datentransfer an die Blockstruktur der externen Dateien angepasst. Beide Maßnahmen beschleunigen die Ein-/Ausgabe um etwa das 100 bis 1000 Fache.

Die `FILE`-Zeiger: Die Ein-/Ausgabefunktionen des Pakets `stdio.h` benötigen einen Zeiger `FILE *fp` auf die `FILE`-Struktur, die zu der jeweiligen Datei gehört (siehe auch nächsten Punkt). Sollte beispielsweise beim Lesen der Puffer leer sein, wird er mittels `fp->bsize = read( fp->fd, fp->buf, BS )` wieder gefüllt; beim Schreiben wird analog verfahren.

Zugriffsrechte auf Dateien: Mit Ausnahme der Standardein- und -ausgabe muss man Dateien immer öffnen, bevor man auf sie zugreifen kann. Dies ähnelt Variablen, die man erst nach ihrer Deklaration verwenden kann. Beim Öffnen einer Datei muss man sich entscheiden, ob man lesen `"r"`, schreiben `"w"` oder verändern `"u"` möchte. Da weitere Details und Kombinationsmöglichkeiten betriebssystemabhängig sind, muss man die Dokumentation gründlich lesen. Ein einfaches Beispiel mit schreibendem Zugriff sieht wie folgt aus:

```
1  #include <stdio.h>
2
3  int main( int argc, char *argv )
4      {
5          FILE *fp;
6          if ((fp = fopen( "test.txt", "w" )) != 0 )
7          {
8              fprintf( fp, "meine erste testnachricht\n" );
9              fclose( fp );
10         }
11     }
```

Sollte die Datei `test.txt` erstellt werden, erscheint in ihr tatsächlich die Zeichenkette `meine erste testnachricht\n`. Obiges Beispiel zeigt auch, dass die Funktion `fopen()` einen Nullzeiger zurückgibt, falls die angegebene Datei nicht geöffnet werden kann.

Gleichzeitiges Verwenden verschiedener Ein-/Ausgabefunktionen: Das Paket `stdio.h` stellt verschiedene Ein- und Ausgabefunktionen zur Verfügung (siehe auch Kapitel 64). Diese können alle gleichzeitig (sozusagen gemischt) verwendet werden. Alle Funktionen benutzen die selben internen Mechanismen, sodass alle Ein- und Ausgaben in der richtigen Reihenfolge gelesen bzw. geschrieben werden.

Gleichzeitiges Verwenden der System Calls: Das wird schon schwieriger. Beim Schreiben auf Dateien kann man dies machen. Nur sollte man *vor* dem Aufruf des `write()` System Calls den zugehörigen `FILE`-Puffer (`fp->buf`) durch den Aufruf `fflush( fp )` zur Ausgabe bringen; die Funktion `fflush( fp )` ruft intern selbstständig den `write()`-System Call auf (siehe obiges Beispiel). Beim Lesen gibt es keine vergleichbare Empfehlung, denn die Dinge, die schon in den internen Puffer `fp->buf` gelesen wurden, kann man *nicht* mehr an das Betriebssystem zurückgeben. Insofern sollte man sich beim Lesen auf eine der beiden Schnittstellen beschränken.

Programmende und Inhalt der `FILE`-Puffer: Sollte das Programm abstürzen und noch etwas in den `FILE`-Puffern sein, so geht deren Inhalt verloren. Ja, die Inhalte sind dann weg. Sollte das Programm regulär beendet werden, werden die Puffer regulär zur Ausgabe gebracht. Das liegt daran, dass am Ende durch die `_init()`-Funktion automatisch die Funktion `exit()` aufgerufen wird. Diese `exit()`-Funktion ruft für alle offenen Dateien von sich aus die Funktion `fflush()` auf, sodass alle zwischengespeicherten Inhalte mittels `write()` zur Ausgabe gebracht werden. Eine kleine Besonderheit bilden die Bildschirmausgaben. Da es sich dabei um Gerätedateien handelt, rufen alle Ausgabefunktionen des Pakets `stdio.h` am Ende immer `fflush()` auf, sodass automatisch alle Inhalte zur Ausgabe gebracht werden. Aber wie gesagt, dies gilt nur für Gerätedateien.

Schnelles „Spulen“ mittels `fseek()`: Wie oben beschreiben, verwaltet das Betriebssystem für jede geöffnete Datei eine aktuelle Lese- bzw. Schreibposition. Mittels des Funktionsaufrufs kann diese Position beliebig verändert werden.

Die Standardeingabe und Standardausgaben: Für die drei Standardkanäle `0`, `1` und `2` gibt es Entsprechungen, die `stdin`, `stdout` und `stderr` heissen. Bei allen drei Variablen handelt es sich um Zeiger auf eine Struktur `FILE`. Die entsprechenden Definition befinden sich meist in der Standardbibliothek `libc` und könnten wie folgt aussehen:

```
1 FILE ftable[ 3 ] = { { 0, 0, 0 }, { 1, 0, 0 }, { 2, 0, 0 } };
2                          // init fileno, bsize, nextc 3 times
3 FILE *stdin  = ftable;
4 FILE *stdout = ftable + 1, *stderr = ftable + 2;
```

Zusammenfassung: In diesem Kapitel haben wir besprochen, wie die `FILE`-Struktur aussieht und wie ein C-Programm mittels der `stdio`-Funktionen mit dem Betriebssystem interagiert. Wichtig ist, dass die `FILE`-Strukturen nicht zum Kernel gehören sondern im Data-Segments des Programms liegen. Man könnte, wenn man wollte, die einzelnen Teile manipulieren. Aber davon raten wir strikt ab, denn das ist den Experten vorbehalten ;-)

Kapitel 64

Die Standard Ein-/Ausgabe Funktionen

In diesem Kapitel versuchen wir endlich, die einzelnen Funktionen und ihre Bedeutungen, Funktionalitäten sowie Rückgabewerte zu erläutern. Dabei gehen wir so vor, dass wir in jedem Kapitel erst die Grundfunktionalität erklären, dann weitere Funktionen nennen, die eine ähnliche Funktionalität aufweisen, und abschließend auf Besonderheiten eingehen.

64.1 Ausgabefunktionen wie `printf()` und `fputc()`

Formatierte Ausgabe: Wie alle Leser jetzt wissen sollten, dient die Funktion `printf()` vor allem der formatierten Ausgabe. Ihr erster Parameter ist eine Zeichenkette. Diese wird als Text interpretiert und ausgegeben. Sollte dabei eine Formatierung vorkommen, die immer mit einem Prozentzeichen anfängt, wird diese durch den Wert des nächsten Parameters ersetzt. Der „Witz“ dabei ist, dass diese Wertersetzung durch Angabe verschiedener Längen formatiert werden kann. Damit lassen sich gut lesbare, tabellarische Ausgaben erstellen.

Rückgabewert: Die Funktion `printf()` gibt die Zahl der ausgegebenen Zeichen als Funktionswert zurück. Beispiel: Die folgende Zeile hat zwei Ausgaben zur Folge:

```
printf( "%d\n", printf( "hi, I am %d\n", 4711 ) );
```

`hi, I am 4711` und `14`, da in der ersten Zeile einschließlich des Zeilenwechsels genau `14` Zeichen ausgegeben wurden.

Weitere Funktionen mit ähnlicher Funktionalität:

`fprintf( FILE *fp, ... ):`
Diese Funktion arbeitet genauso wie die Funktion `printf()`, nur dass man als erstes Argument explizit einen File Pointer `fp` übergeben muss. Es gilt:

`printf( <Parameter> )` $\Leftrightarrow$ `fprintf( stdout, <Parameter> )`

`sprintf( char *str, ... )`

Diese Funktion ist funktional identisch mit:

`sprintf( str, <Parameter> )` ⇔ `fprintf( fp, <Parameter> )`,

außer dass nicht auf einen File Pointer `fp` sondern direkt in den Puffer `str` vom Typ `char *` geschrieben wird.

`fputc( int c, FILE *fp ):`

Diese Funktionen bewirkt das gleiche wie:

`fputc( c, fp )` ⇔ `fprintf( fp, "%c", c )`,

nur dass sie den ASCII-Code des ersten Parameters `c` zurückgibt. Beispiel: Der Rückgabewert von `fputc( 'x', stdout )` ist 120, da `'x'` genau diesen Wert hat.

`fputs( char *str, FILE *fp ):`

Diese Funktion gibt eine ganze Zeichenkette `str` aus und ist damit identisch zu:

`fputs( c, fp )` ⇔ `fprintf( fp, "%s", str )`,

wobei nur im Fehlerfalle ein `EOF` zurückgegeben wird, sonst wird eine nicht negative Zahl zurückgegeben.

`putchar( int c )`, `putc( int c, FILE *stdout ):`

Außer, dass es sich bei diesen beiden Funktionen um ein Makro handeln kann, gilt:

`putchar( c )` ⇔ `putc( c, stdout )` ⇔ `fputc( c, fp )`,

Grundsätzlich kann man alle obigen Ausgabefunktionen gemischt verwenden wie man will. Alle Funktionen schreiben in den Puffer, der zur `FILE`-Struktur gehört, sodass in allen Fällen die Reihenfolge aller Ausgaben erhalten bleibt.

Besonderheit: gleichzeitige Verwendung des System Call `write()`: Alle obigen Ausgabefunktionen kann man sogar gleichzeitig mit dem System Call `write()` verwenden. Man sollte aber unbedingt zuvor die sich möglicherweise im `struct FILE`-Puffer befindlichen Daten mittels `fflush( fp )` tatsächlich zur Ausgabe bringen. Die Aufrufreihenfolge lautet also: `fprintf( fp, ... )` ⇒ `fflush( fp )` ⇒ `write( fileno( fp ) )`.

Besonderheit: Programmende: Am Ende des Programms ist etwas „Vorsicht“ geboten. Schreibt man auf den Bildschirm, also bei Euch immer auf `stdout` oder `stderr`, werden die Ausgaben trotz Zwischenspeicherung am Ende des verantwortlichen Funktionsaufrufs automatisch mittels `fflush()` tatsächlich zur Ausgabe gebracht. Sollte aber die Ausgabe durch Öffnen einer ausgewählten Datei oder durch Umlenken in eine Datei gehen, werden alle Ausgaben solange zwischengespeichert, bis der `FILE`-Puffer voll ist. Da aber die `FILE`-Strukturen zum Programm und nicht zum Kernel gehören, kann der Kernel die im Puffer befindlichen Zeichen nicht zur Ausgabe bringen; sie gehen schlichtweg verloren. Daher sollte man beim Schreiben auf Dateien am Programmende entweder `fflush()` oder `fclose()` (jeweils auch für `stdout` und `stderr`) aufrufen.

64.2 Eingabefunktionen wie `scanf()` und `fgetc()`

Vorsorglich sei nochmals wiederholt, dass `scanf()` und die anderen in `stdio.h` deklarierten Eingabefunktionen niemals direkt aus einer Datei sondern immer aus dem Puffer der `FILE`-Struktur lesen. Dieser Puffer wird von diesen Funktionen bei Bedarf durch Aufruf des System Calls `read()` nachgeladen.

Das Sonderzeichen `EOF`: In der Datei `stdio.h` gibt es folgende Konstantendefinition: `#define EOF -1` Im Falle eines Fehlers oder bei Erreichen des Dateiendes geben einige der Lesefunktionen diesen Wert zurück.

„Normale“ Eingabe: Bisher haben wir die Funktion `scanf()` zur Eingabe einzelner Parameter verwendet. Beispiel: mittels `scanf( "%d", & i )` können wir eine ganze Zahl einlesen und der Variablen `i` zuweisen. Bei der Eingabe von Zahlen überliest `scanf()` alle vorangestellten Leerzeichen, Tabulatoren und Zeilenwechsel. Natürlich kann man im ersten Parameter auch mehrere Formatierungen angeben, sodass man mit einem Funktionsaufruf gleich mehrere Parameter lesen kann. Beispiel: `scanf( "%d%c", & i, & c )`

Rückgabewert: Die Funktion `scanf()` gibt die Zahl der erfolgreich konvertierten Argumente zurück. Diese kann naturgemäß von `0` bis zur Zahl der Argumente variieren. Ist der Rückgabewert kleiner als die Zahl der erwarteten Argumente, ist offensichtlich ein Lesefehler aufgetreten. Dies geschieht immer dann, wenn das von `scanf()` bearbeitete aktuelle Zeichen nicht zum aktuellen Format passt. Beispiel: `scanf( "%d", & i )` und die Eingabe `abc`. In diesem Fall ist der Buchstabe `'a'` keine Zahl. Da keine Konvertierung stattfindet, wird eine `0` zurückgegeben und das Zeichen `'a'` *nicht* verarbeitet. Entsprechend wird ein erneuter Aufruf von `scanf( "%d", & i )` zum selben Resultat führen. Die Behandlung derartiger Fälle behandeln wir in Abschnitt 64.3 Bei Erreichen des Dateiendes gibt `scanf()` den Wert `EOF` zurück. Ein fehlertolerantes Programm würde wie folgt aussehen:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4      {
5          int i;
6          switch( scanf( "%d", & i ) )
7          {
8              case   1: printf( "Eingelesene Zahl: %d\n", i );
9                        break;
10             case   0: printf( "Keine Zahl gefunden\n" ); break;
11             case EOF: printf( "Dateiende\n" ); break;
12         }
13     }
```

Formatierte Eingabe: Eine wesentliche Eigenschaft der Funktion `scanf()` ist, dass es die Eingabe „genau“ mit der Formatierung abgleicht, in der sich auch ganz normale Zeichen befinden können. Beispiel: Der Aufruf `scanf( "A%d", & i )` gibt nur dann eine `1` zurück,

wenn vor der Zahl auch tatsächlich das Zeichen `'A'` erscheint. Sollte der Nutzer `B123` eingeben, kommt eine `0` zurück und die aktuelle Eingabeposition bleibt (in diesem Beispiel) unverändert. Bei Interesse einfach mal die `manpage` lesen; Windows Nutzer schauen am besten unter `man7.org`.

Eingabe mittels `fgetc()`: Die Funktion `fgetc( fp )` liest das nächste und nur das nächste Zeichen vom angegebenen File Pointer `fp`. Nun hat diese Funktion aber ein Problem: Es gibt 256 gültige Zeichen, die mit genau einem Byte (acht Bits) kodiert werden können; für ein weiteres Zeichen ist einfach kein Platz. Aber neben den 256 möglichen Zeichen muss `fgetc()` auch das Ende der Datei signalisieren. Aus diesem Grund erhielt die Konstante `EOF` den Wert `-1` und `fgetc()` den Typ `int`. Beispiel: Kopieren der Standard-Eingabe auf die Standard-Ausgabe:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4     {
5         int c;
6         while((c = fgetc( stdin )) != EOF )
7             fputc( c, stdout );
8     }
```

Weitere Funktionen mit ähnlicher Funktionalität:

`fscanf( FILE *fp, ... ):`
Diese Funktion arbeitet genauso wie die Funktion `scanf()`, nur dass man als erstes Argument explizit einen File Pointer `fp` übergeben muss. Es gilt:

`scanf( <Parameter> )` $\Leftrightarrow$ `fscanf( stdin, <Parameter> )`

`sscanf( char *str, ... ):`
Diese Funktion ist funktional identisch mit:

`sscanf( str, <Parameter> )` $\Leftrightarrow$ `fscanf( fp, <Parameter> )`,

ausser dass nicht von einem File Pointer `fp` sondern aus dem Puffer `str` gelesen wird.

`fgetc( FILE *fp ):`
Diese Funktionen ist mit Ausnahme des `EOF`-Falls funktional identisch mit

`c = fgetc( fp )` $\Leftrightarrow$ `fscanf( fp, "%c", & c )`.

`fgets( char *str, int size, FILE *fp ):`
Diese Funktion liest eine ganze Zeichenkette ein und legt sie im Puffer `str` ab. `fgets()` überprüft beim Einlesen die Länge `size` des zur Verfügung gestellten Puffers `str`. Das Einlesen wird bei Antreffen des Dateiendes, eines Zeilenwechsels `\n` oder dem Erreichen der Puffergröße beendet. Der Rückgabewert ist entweder die Adresse des übergebenden Puffers (erfolgreiches Lesen) oder 0 (Fehler).

`gets( char *str )`:
Diese Funktion ist funktional identisch mit:

`gets( fp )` ⇔ `scanf( "%s", str )`

Da beide Varianten nicht überprüfen, ob über das Ende des Puffers `str` hinaus geschrieben wird, können schwer zu findende Fehler auftreten. Daher sind beide Varianten unsicher und sollten in keinem Fall verwendet werden.

`getchar()`, `getc( FILE *fp )`:
Es gilt:

`getchar()` ⇔ `getc( stdin )` sowie `getc( fp )` ⇔ `fgetc( fp )`,

wobei es sich bei beiden Funktionen um Makros handeln kann.

64.3 Reaktion auf Eingabefehler

Es ist schwer, hier ein allgemeingültiges Kochrezept zu vermitteln. Eine Lösungsmöglichkeit besteht darin, jede Eingabezeile zeichenweise einzulesen und die Konvertierungen selbst durchzuführen. Aber das ist meistens „mit Kanonen auf Spatzen geschossen“ und für die meisten von Euch noch zu schwer.

Ein zweiter Lösungsansatz besteht darin, sich genau anzuschauen, was eigentlich passiert. Nehmen wir an, wir hätten die C-Anweisung `scanf( "%d", & i )` und der Nutzer gibt zwei Leerzeichen und ein `'x'` ein. `scanf()` würde wie üblich die beiden Leerzeichen überlesen und das `'x'` antreffen. Da aber dieses `'x'` zu keiner Zahl gehört, würde `scanf()` abbechen, die Variable `i` unverändert lassen und eine `0` zurückgeben. Auch ein erneuter Aufruf von `scanf( "%d", & i )` würde dieses „fehlerhafte“ `'x'` antreffen, sodass das Programm ohne Gegenmaßnahmen in eine Endlosschleife laufen würde. Entsprechend ist in solchen Fällen folgende Formulierung unbrauchbar:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4     {
5        int i;
6        do { scanf( "%d", & i );
7             printf( "i= %d\n", i );
8           } while( i >= 0 );
9     }
```

Ein erster Schritt zur Fehlerbehandlung sieht wie folgt aus:

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

```
 3 int main( int argc, char **argv )
 4     {
 5        int i;
 6        do switch( scanf( "%d", & i ) )
 7           {
 8             case   1: printf( "i= %d\n", i ); break;
 9             case   0: break;              // error handling
10             case EOF: i = -1; break;     // terminates loop
11           } while( i >= 0 );
12     }
```

Eine konkrete Möglichkeit zur Fehlerbehandlung könnte sein, dass man alles bis zum Ende der Zeile überliest. Dies könnten wir einfach wie folgt erreichen:

```
10                          do c = getc(stdin);
11                          while( c != '\n' && c != EOF);
```

Wenn wir dies in unser Programm integrieren, erhalten wir folgendes:

```
 1 #include <stdio.h>
 2
 3 int main( int argc, char **argv )
 4     {
 5        int i, c;
 6        do switch( scanf( "%d", & i ) )
 7           {
 8             case   1: printf( "i= %d\n", i ); break;
 9             case   0: printf( "Eingabe ist keine Zahl\n" );
10                       do c = getc(stdin);
11                       while( c != '\n' && c != EOF);
12                       break;              // error handling
13             case EOF: i = -1; break;     // terminates loop
14           } while( i >= 0 );
15     }
```

Kapitel 65

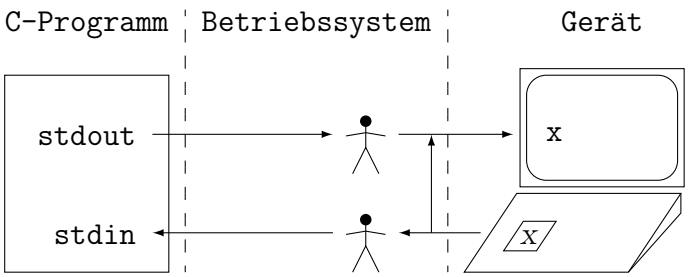
Besonderheiten der Terminaleingabe

Es ist kaum zu glauben, aber die Terminaleingabe ist sehr speziell. *„Weil davor ein Dumm-User sitzt ;-) ?“* Nein, wir meinen nicht Dich ;-). In diesem Minikapitel geht es um die technische Konstruktion der Terminalein- und -ausgabe. Diese ist aufgrund ihrer Konstruktion sehr speziell, wie die folgende Skizze illustriert:

Technische Sicht:

- Zwei Geräte:
  Tastatur zur Eingabe
  Terminal zur Ausgabe
- Zwei Treiberkomponenten:
  Eine für die Eingabe
  Eine für die Ausgabe
- Zwei File Descriptoren:
  `stdin` für die Eingabe
  `stdout` für die Ausgabe

Konzeptuelle Sicht:



Da technisch gesehen ein Terminal eigentlich aus zwei eigenständigen Geräten besteht, ist es vielleicht gar kein so großes Wunder, dass wir sowohl einen Ein- als auch einen Ausgabekanal verwenden. Diese beiden Kanäle müssen wir strickt voneinander unterscheiden. Entsprechend haben wir auch zwei Treiberkomonenten, eine zum Lesen und eine zum Schreiben, die wir in obiger Skizze als kleine Männchen dargestellt haben.

Ist das Terminal tatsächlich ein echtes Gerät (und nicht umgelenkte Dateien) handelt es sich bei den Männchen um „Zwillinge“, die eng miteinander kommunizieren. Wie in obigem Bild skizziert ist, nimmt der „Lese-Zwilling“ jeden Tastendruck und damit jedes Zeichen entgegen und leitet es direkt an seinen Ausgabepartner weiter. Dieser stellt es sogleich auf dem Bildschirm dar, damit wir sehen, was wir tippen.

Im Falle von Terminals zeigen die beiden Treiber-Männchen noch ein weiteres „merkwürdiges“ Verhalten: Das Eingabemännchen leitet zwar alles an seinen „Zwilling“ weiter, doch

behält ansonsten alles für sich. Es leitet keines der eingegebenen Zeichen an das Betriebssystem oder unser Programm weiter, sondern bunkert einfach alles. Entsprechend merkt unser C-Programm nichts von den getätigten Eingaben. Ein `scanf()` oder `read()` System Call müsste warten, denn das Eingabemännchen rückt die Eingaben einfach nicht heraus, obwohl sie bereits im internen `FILE`-Puffer vorliegen. *Erst* durch Betätigen der Enter-Taste kann unser C-Programm auf die lokal gespeicherten und bereits auf dem Bildschirm angezeigten Zeichen zugreifen.

Jetzt sind wir in der Lage, das Verhalten von Programmen zu verstehen, die abwechselnd mehrere Aus- und Eingaben veranlassen. Schauen wir auf folgendes Programm:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4     {
5         int i, j;
6         printf( "Bitte i eingeben: " ); scanf( "%d", & i );
7         printf( "Bitte j eingeben: " ); scanf( "%d", & j );
8         printf( "i= %d j= %d\n", i, j );
9     }
```

Nach der ersten Ausgabe (der Eingabeaufforderung für `i`) will unser Programm einen Wert für die Variable `i` einlesen. Nehmen wir an, wir geben gleich zwei Zahlen beispielsweise `4711` und `007` ein. Dann passiert folgendes: Das Eingabemännchen liest die erste Zahl (`4711`), leitet sie zwar zur Ausgabe weiter doch behält sie ansonsten für sich, da wir die Enter-Taste noch nicht gedrückt haben. Anschließend ließt das Eingabemännchen auch die zweite Zahl (`007`) und die Enter-Taste (`\n` in C-Programmen). Nun können die Eingaben von unserem Programm gelesen werden. Das erste `scanf()` erhält jetzt die erste Zahl (`4711`) und ist zufrieden. Die zweite Zahl `007` bleibt noch im Treiber zurück, da unser `scanf()` ja nur eine Zahl haben wollte. Nach der nächsten Programmausgabe will unser Programm einen Wert für die Variable `j` einlesen. Da diese bereits im Treiber vorhanden ist, wird das `scanf()` sofort zufriedengestellt, *ohne* erst auf eine weitere Nutzereingabe zu warten. Anschließend wird unser C-Programm seine dritte Ausgabe (Zeile `8`) ohne Verzögerung abarbeiten.

Das hier beschriebene spezielle Verhalten von Terminaltreibern nennt man auch lokales Editieren und erfüllt einen wichtigen Zweck. Es erklärt, was beim Drücken der `Back Space` Taste passiert: Das Eingabemännchen entfernt das zuletzt eingegebene Zeichen aus seinem internen Puffer und veranlasst das Ausgabemännchen dieses Zeichen vom Bildschirm zu entfernen. Gäbe es diese Funktionalität nicht, müssten wir sie selbst implementieren, was nicht trivial ist. Beispielsweise schaltet die Shell das Echo und das lokale Editieren mittels `stty` (Kommando und System Call) ab, um Dinge wie *file name completion* zu realisieren. Doch eine eingehende Behandlung dieser Funktionalität ist für den Programmieranfänger zu schwer.

Kapitel 66

Für Interessierte: Die Ein-/Ausgabe aus Sicht des Betriebssystems

In den zurückliegenden Kapiteln haben wir versucht, einen ungefähren Überblick über die Ein-/Ausgabe zu geben. Dieses Thema ist inherent tierisch schwer! Man könnte leicht ein paar hundert Seiten eines Lehrbuchs damit füllen. Entsprechend haben wir sehr stark vereinfacht. In diesem Kapitel wollen wir ein wenig intensiver auf die Interna des Betriebssystem eingehen. Aber Achtung, dieses Kapitel ist nur für wirklich Interessierte! Der Inhalt ist schwer, erfordert ein mehrmaliges Lesen und ist daher nicht Prüfungsgegenstand.

Solange ein Programm nur Rechenoperationen ausführt, können Compiler und CPU die Arbeit alleine erledigen. Aber die Ein- und Ausgabe muss die Hardware ansprechen, wofür vor allem das Betriebssystem verantwortllich ist. Wer verstehen will, wie die Ein- und Ausgabe funktioniert, muss auch in groben Zügen verstehen, was das Betriebssystem macht.

Bevor wir anfangen, müssen wir uns darüber im klaren werden, ob wir ein Programm für einen PC oder einen Mikrokontroller entwickelen. Auf einem PC läuft ein Betriebssystem, das für die Verwaltung der Hardware zuständig ist. Diese Betriebssystemschicht sorgt dafür, dass unsere Ein- und Ausgaben über spezielle Treiber von und zur Hardware transportiert werden. Auf einem Mikrokontroller gibt es in der Regel kein Betriebssystem, weil entweder keines notwendig ist oder hierfür die Ressourcen nicht ausreichen. Auf vielen Mikrokontrollern müssen wir alles selbst programmieren, da beispielsweise ein komplexes `printf()` nicht oder nur eingeschränkt zur Verfügung steht. Typische Mikrokontrolleranwendungen sind Waschmaschinen, Kaffeeautomaten und mp3-Spieler.

66.1 Historischer Hintergrund

„Schon wieder so'n Asbach-Zeugs?“ Ja, kennt man die historischen Hintergründe, versteht man auch die nachfolgenden Konzepte wesentlich leichter, denn dann kennt man deren Intentionen. Also, fangen wir vorne an. Damals, als das Betriebssystem Unix entwickelt

wurde, merkten die Profi-Entwickler, dass das Programmieren in Assembler viel zu ineffizient ist. In ihrem Bestreben nach einem neuen, klaren, gut funktionierenden Betriebssystem entstand ganz nebenbei die Programmiersprache C. Die Jungs waren einfach super Profis.

Neben der zeitlichen Koinzidenz hat die gemeinsame Entwicklung von Unix und C weitere Konsequenzen: einige Konzepte des Betriebssystem finden sich auch im Sprachdesign wieder. Hierzu zählt eben auch die Ein-/Ausgabe Funktionalität, die im englischen auch als Input/Output Subsystem bezeichnet wird.

Eine wesentliche, recht revolutionäre Designentscheidung war, dass in Unix alles Dateien sind. Unter „alles“ sind nicht nur Quelltexte und Dokumente zu verstehen sondern auch übersetzte Programme, laufende Programme (diejenigen, die gerade von der CPU abgearbeitet werden), Directories, Plattenlaufwerke, Bildschirme, Tastaturen, einfach alle Geräte. Ja, alle Geräte sind Dateien. Entsprechend merkt man in seinem C-Programm nicht, ob die Daten gerade aus einer Datei kommen, oder ob sie ein Nutzer über die Tastatur eingibt. Diesen Unterschied herauszufinden ist sogar mit Aufwand verbunden. Die Unterschiede der einzelnen Dateien und Geräte werden durch die Treiber und weitere untere Schichten gekapselt.

66.2 Die Aufgaben des Betriebssystems

Bereits seit Kapitel 4 haben wir immer wieder vom Betriebssystem erzählt. Aber was genau macht eigentlich dieses Betriebssystem? In „modernen“ Systemen ist das Betriebssystem ebenfalls ein Stück Software, dass ebenfalls im Arbeitsspeicher zu finden ist. Aber das Betriebssystem ist kein Programm, das eigenständig für sich etwas macht. Im Grunde genommen ist das Betriebssystem eine Ansammlung von Funktionen, die je nach „Auftragslage“ eine bestimmte „Dienstleistung“ für eines der anderen Programme liefert. Während der Diensterbringung ist die Funktion quasi Teil des Programms, anschließend liegt sie wieder ohne weitere Zugehörigkeit im Arbeitsspeicher herum.

Die erste wesentliche Aufgabe des Betriebssystem (eigentlich der Menge der einzelnen Betriebssystemfunktionen) ist die Kontrolle alle Hardware-Komponenten einschließlich CPU und RAM. In diesem Sinne stellt das Betriebssystem dem laufenden Programm eine Abstraktionsschicht der Hardware zur Verfügung, die das Ansprechen der Hardware deutlich vereinfacht.

Die zweite wesentliche Aufgabe des Betriebssystem hat direkt mit den Nutzern zu tun: Es muss ausreichend Sicherheitsmechanismen zur Verfügung stellen, sodass sich die Nutzer und ihre Programme nicht gegenseitig behindern. Hierzu gehören auch Sicherheitsmechanismen, die unerlaubte Datenmanipulationen und Datenzugriffe unterbinden.

Bei den ersten Versionen von Windows waren diese Sicherheitsmechnismen nicht im Fokus, sodass ein Programm die internen Daten eines anderen Programms lesen und sogar verändern konnte, was für viele Leute ein sehr interessantes „Angriffsziel“ darstellte. Un-

ix hingegen war von Anfang an als Multi-User, Multi-Tasking Betriebssystem[1] ausgelegt, sodass starke Sicherheitsmechanismen schon in den Entwurf integriert wurden. Dies ergab aber einen etwas komplexeren Entwurf des Betriebssystems, den wir uns im nächsten Abschnitt etwas genauer ansehen.

66.3 Vom Gerät zum Programm

Auf der nächsten Seite haben wir die westlichen Komponenten skizziert, die seitens des Betriebssystems bei der Ein- und Ausgabe mitwirken. Dieses Bild ist sehr komplex. Im Folgenden werden wir die einzelnen Komponenten der Reihe nach kurz besprechen. Dabei gehen wir so vor, dass wir uns mal von oben, mal von unten nähern, bis beide Ansätze in der Mitte zusammentreffen.

Unser Programm:
Wir stellen uns einfach mal vor, einer der Doktoranden ist dabei, den ersten Entwurf der C-Klausur zu erstellen. Hierzu tippt er `gedit Klausur.txt` in seinen Rechner. Daraufhin wird das Programm `gedit` gestartet, das seinerseits den aktuellen Stand der Datei `Klausur.txt` in den Arbeitsspeicher einliest.

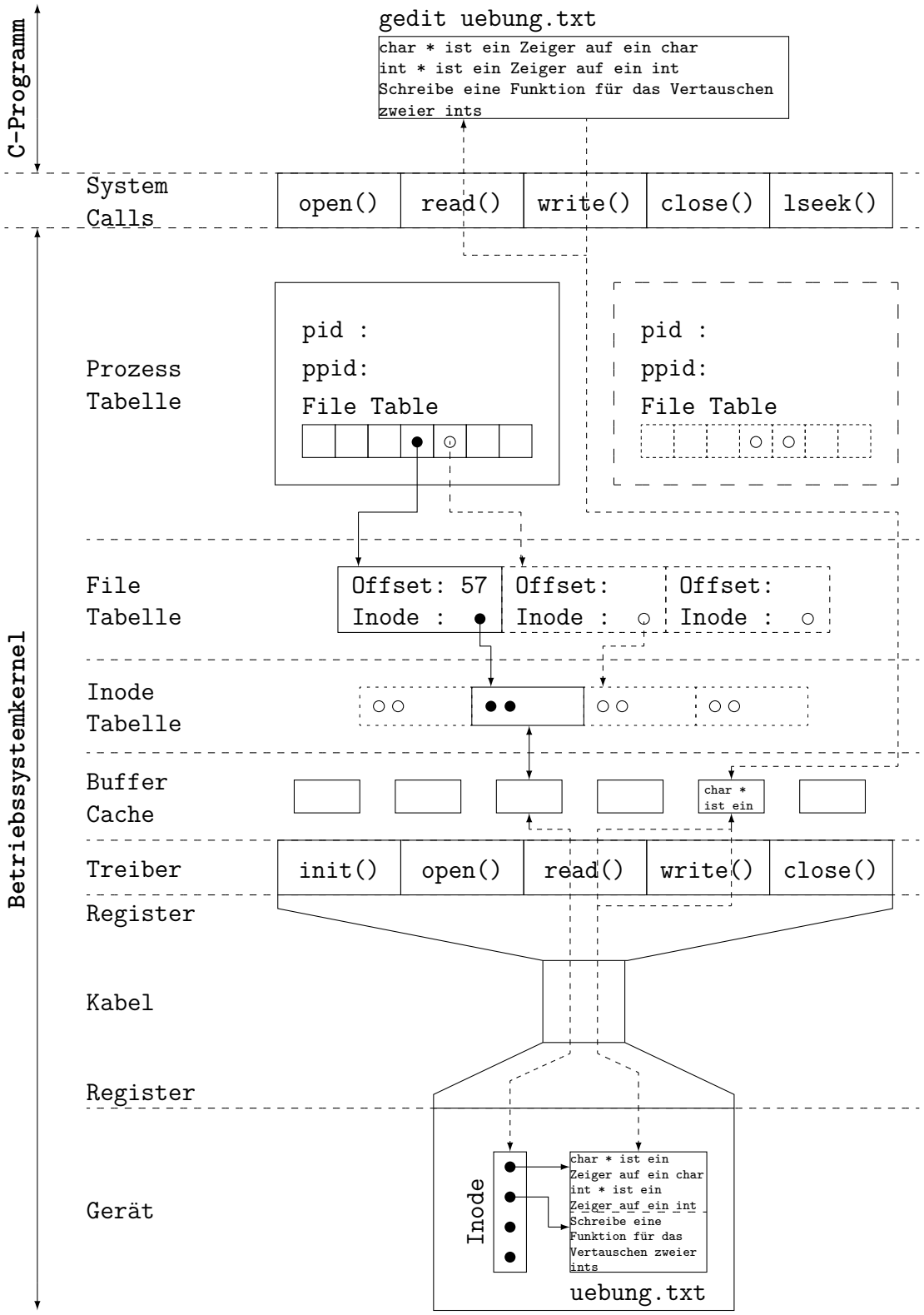
Die Dateien auf dem Datenträger: Ganz unten ist die Hardware, die bei Massendaten in der Regel Plattenlaufwerke oder moderne SSDs sind. Auf diesen Datenträgern befinden sich die eigentlichen Daten (auch alle Programme, da ja alles und jedes eine Datei ist) in Dateien, die aus einzelnen Datenblöcken à 512, 1024 oder 2048 Byte bestehen. Zu jeder Datei gehört eine Inode (die Inode, *fem.*), die die Struktur dieser Datei beschreibt[2]. Neben dem Eigentümer gehört dazu die Größe der Datei sowie die Position der einzelnen Datenblöcke auf dem Datenträger.

Kabel, Register und Geräte:
Alle Geräte sind irgendwie mittels Kabel direkt oder indirekt an die CPU angeschlossen. Manchmal sieht man diese Kabel nicht, aber sie sind da. Selbst bei USB ist es so, sie befinden sich im Stecker und auf den Adapterplatinen. Im Regelfall ist es so, dass die Geräte mittels eines weiteren Mikrokontrollers (Prozessor) gesteuert werden. Zur Kommunikation beider Seiten (Controller und (Haupt-) CPU) dienen Register, die sich auf beiden Seiten der Kabel befinden. Durch diesen Kommunikationsweg wird beispielsweise erreicht, dass ein Plattenlaufwerk einen vorher spezifizierten Datenblock liest.

Treiber:
Oberhalb der Kabel und Register befinden sich die Hardware-Treiber. Sie sorgen

[1] Früher haben tatsächlich viele Leute gleichzeitig per Terminal an einem Unix Rechner gearbeitet und dabei ohne Probleme gleichzeitig ihre Diplom- und Doktorarbeiten fertiggestellt.

[2] Der Name der Datei ist weder in der Inode noch in der Datei selbst abgelegt. Die Verbindung aus Name und Inode wird durch einen entsprechenden Eintrag in der Directory hergestellt. Bei Fragen helfen die Betreuer gerne weiter.


C-Programm
Betriebssystemkernel
gedit uebung.txt
char * ist ein Zeiger auf ein char
int * ist ein Zeiger auf ein int
Schreibe eine Funktion für das Vertauschen
zweier ints
System
Calls
open()
read()
write()
close()
lseek()
Prozess
Tabelle
pid :
ppid:
File Table
pid :
ppid:
File Table
File
Tabelle
Offset: 57
Inode :
Offset:
Inode :
Offset:
Inode :
Inode
Tabelle
Buffer
Cache
char *
ist ein
Treiber
init()
open()
read()
write()
close()
Register
Kabel
Register
Gerät
Inode
char * ist ein
Zeiger auf ein char
int * ist ein
Zeiger auf ein int
Schreibe eine
Funktion für das
Vertauschen zweier
ints
uebung.txt

dafür, dass durch Absetzen der richtigen Kommandos in Richtung der Geräte diese genau das machen, was der Kernel (Betriebssystemkern) will. Wie die Abbildung zeigt, gehören das Initialisieren, Öffnen und Schließen sowie das Lesen und Schreiben zu den typischen Funktionen dieser Treiberschnittstelle. Eine wesentliche Entwurfsentscheidung im Rahmen des Unix-Projektes war, diese Treiberschnittstelle für alle Geräte identisch auszulegen. Dadurch muss sich das Betriebssystem nicht darum kümmern, welches Gerät es anspricht; es ruft einfach die entsprechenden Treiberfunktionen auf. Dies bedeutet auch, dass für jedes neues Gerät, dessen Schnittstelle sich von denen der anderen Geräten unterscheidet, ein neuer Satz Treiberfunktionen erstellt werden muss.

Buffer Cache:
Aufrufe der Gerätetreiber sind sehr kostspielig (im Sinne von Rechen- und Wartezeit)[3]. Für die Optimierung des Gesamtsystems werden die gelesenen (und geschriebenen) Datenblöcke im Buffer Cache zwischengespeichert. Dieser Buffer Cache ist im Arbeitsspeicher und damit viel dichter an der CPU, was zu einer wesentlich höheren Zugriffsgeschwindigkeit führt, als es bei externen Geräten möglich ist. Dieser Ansatz basiert auf der Beobachtung, dass sehr häufig einige Datenblöcker innerhalb kurzer Zeit wiederholt gelesen werden. Wenn man beispielsweise das Kommando `ls` zwei Mal nacheinander ausführen will, wird es beim ersten Mal vom Plattenlaufwerk, beim zweiten Mal aus dem Buffer Cache geladen, was deutlich schneller ist.

Inode Tabelle:
Oberhalb des Buffer Cache befindet sich die Inode-Tabelle, in der sich die Inodes (die Strukturbeschreibungen) aller momentan offenen Dateien befinden. Mit anderen Worten: Wann immer ein Programm eine Datei öffenen will, wird die entsprechende Inode in die Inode-Tabelle geladen, sodass der Kernel weiß, wie groß diese Datei ist, wo sich welche Datenblöcke befinden usw. An dieser Stelle sei für den wirklich Interessierten erwähnt, dass das Laden einer Inode das vorherige Lesen eines Datenblocks erfordert, da sich ja die Inodes auch auf dem Gerät befinden, also Bestandteil einer Datei sind.

Prozess Tabelle:
Weiter oben befindet sich die Prozesstabelle. Hier ist für jedes aktive Programm, das innerhalb des Betriebssystem als Prozess bezeichnet wird, ein Tabelleneintrag zu finden, den man unter Unix auch Process Control Block (PCB) nennt. Neben allerlei Verwaltungsinformationen speichert der Process Control Block die Id des Prozesses (Process ID (PID)) und die Id des Vaterprozesses (Parent Process ID (PPID)).

Ferner hat der Process Control Block eine eigene File-Tabelle. Von dieser prozessspezifischen File-Tabelle geht es zu einer systemweiten File-Tabelle, von wo aus es zu

[3] Beispielsweise dauert das Positionieren des Schreib-/Lesekopfes eines Plattenlaufwerkes sowie das Lesen des entsprechenden Datenblocks ungefähr zehn bis 50 Millisekunden. Jeder kann sich ausrechnen, wie viele Rechenoperationen eine CPU bei einer Taktfrequenz von 2 GHz in dieser Zeit ausführen kann.

den Inodes geht. Dieser etwas umständlich anmutende Ansatz über zwei File- und einer zusätzlichen Inode-Tabelle hat seine Gründe, deren Diskussion aber den Rahmen sprengen würde. Für die Interessierten sein die Originalliteratur [1] empfohlen.

Wir sollten aber vier Dinge festhalten: Erstens, den Index innerhalb der dem Prozess gehörenden File-Tabelle nennt man *File Descriptor* (FD). Zweitens, jedes Element dieser File-Tabelle besteht aus einem Zeiger, der in die globale File-Tabelle zeigt und den der Kernel bei allein Ein- und Ausgaben auswertet. Drittens, immer wenn wir eine Datei öffnen, wird der nächste freie Eintrag dieser File-Tabelle verwendet; dieser Index wird File Descriptor (FD) genannt und uns als Ergebnis zurückgegeben. Viertens, auf die Inhalte der File-Tabelle hat man keinen direkten Zugriff; dieser erfolgt über System Calls, die die File Descriptoren als Parameter benötigen.

File Tabelle:
Die File Tabelle stellt die Verbindung der File Descriptoren (innerhalb des Process Control Blocks) zu den Dateien her, in dem sie einen Verweis auf die entsprechende Inodes enthält. Zusätzlich ist dort die aktuelle Position (genannt Offset) innerhalb der offenen Datei abgelegt. Dadurch weiß der Kernel immer, wo er gerade lesen oder schreiben soll.

Um das Zusammenspiel aller Komponenten etwas zu erläutern, besprechen wir kurz einen kleinen Anwendungsfall. Wir gehen davon aus, dass der Editor `gedit` bereits gestartet ist und die Datei `Klausur.txt` öffnen will.

1. Über die Directory-Struktur findet der Kernel heraus, wo sich die Inode der Datei `Klausur.txt` befindet. Der Kernel würde diesen Datenblock über den `read()`-Gerätetreiber in den Buffer Cache lesen und von dort die Inode in die Inode Tabelle kopieren.

2. Der Kernel würde sich die Inode anschauen und anschließend die zur Datei gehörenden Datenblöcke wiederum über den `read()`-Gerätetreiber in den Buffer Cache lesen, sofern sie dort noch nicht vorhanden sind. Im Zuge dieser Lesevorgänge würde der Kernel regelmäßig den Offset der Datei `Klausur.txt` in der File Tabelle aktualisieren, damit er immer den richtigen Datenblock liest.

3. Vom Buffer Cache gelangen die Datenblöcke der Datei `Klausur.txt` in den Datenbereich des Programms `gedit`, das sich ebenfalls im Arbeitsspeicher befindet. Dieser Arbeitsschritt erfordert ein Umkopieren innerhalb des Arbeitsspeichers vom Buffer Cache zum Nutzerprogramm. Auf der anderen Seite werden durch diesen Arbeitsschritt die belegten Datenblöcke des Buffer Caches wieder frei und können für andere Aufgaben genutzt werden.

Diese Arbeitsschritte werden mehr oder weniger bei jedem Zugriff auf einen (externen) Datenträger abgewickelt.

66.4 Der System Call `lseek()`

Eine Bemerkung sollten wir noch in Richtung des System Calls `lseek()` machen. Mit diesem System Call kann man die aktuelle Position innerhalb einer Datei beliebig verändern. Sofern alle Parameter korrekt sind, wird durch diesen System Call einfach der Offset in der File Tabelle verändert. In Unix (und damit auch Linux) kann beim Schreiben über das aktuelle Ende der Datei hinaus gegangen werden. In diesem Fall werden entsprechende Null-Bytes in die Datei eingefügt. Ferner sollte man wissen und beachten, dass `lseek()` nur auf richtige Dateien anwendbar ist. Sollte der übergebene File Descriptor auf ein Gerät, also die Tastatur oder der Bildschirm, verweisen, wird der System Call `lseek()` mit einer Fehlermeldung zurückgewiesen; ein Neupositionieren von Tastatur und Bildschirm ist also nicht möglich, was irgendwie auch klar sein sollte.

66.5 Umlenken der Standard Ein- und Ausgabe

Nun können wir auch recht einfach verstehen, was beim Testen unserer Übungsprogramme passiert. Wenn wir mittels der Shell (Kommandoeingabe) ein Programm, beispielsweise `uebung-4711` (`uebung-4711.exe`), aufrufen, sind die ersten drei File Descriptoren mit der Tastatur (Descriptor 0) und dem Bildschirm (Descriptoren 1 und 2) verbunden.

Im Übungsbetrieb haben wir gelernt, dass wir die Eingabe wie folgt umlenken können: `uebung-4711 < test-daten`. In diesem Fall führen die Shell und der Kernel *vor* dem eigentlichen Programmstart einige Verwaltungsarbeiten durch. Als erstes wird die Datei `test-daten` geöffnet, wodurch entsprechende Einträge in der Inode- sowie der globalen File-Tabelle erzeugt werden. Anschließend wird der File Descriptor 0 von der Tastatur-Inode auf die neue Datei-Inode umgelegt. Erst jetzt wird unser Testprogramm `uebung-4711` tatsächlich auch gestartet. Es liest weiterhin von File Descriptor 0 und merkt nicht, dass es nicht mehr von der Tastatur sondern aus der Datei `test-daten` liest, da es nichts von den Änderungen in den diversen Tabellen mitbekommen hat. Eigentlich recht *smart* :-)

66.6 Zusammenfassung

In diesem Kapitel haben wir gesehen, wie der Kernel mit Hilfe verschiedener Tabellen und Treiber in der Lage ist, Datenblöcke zwischen den Anwendungen und Datenträgern zu transferieren. Da die Kernel-Datenstrukturen für normale Programme unerreichbar sind, stellt der Kerne einige System Calls zur Verfügung. Durch diese Schnittstelle ist der Kernel in der Lage, seine internen Datenstrukturen auch bei fehlerhaften Programmaufrufen konsistent zu halten. Die oben erwähnten System Calls erwarten einen File Descriptor, der auf einen Eintrag in die File Tabelle des Process Control Blocks verweist. Von hier geht es über die File Tabelle zur Inode Tabelle und von da weiter zu den Geräten, auf denen sich die eigentlichen Daten befinden. Die aktuelle Position `offset` innerhalb einer Datei verwaltet der Kernel in einem Element eines `struct` innerhalb der File Tabelle. Auf diese Position

hat man keinen Zugriff; der Kernel passt sie aber bei jeder Lese- und Schreiboperation automatisch an. Allerdings kann man bei *regulären* Dateien die aktuelle Position mittels des System Calls `lseek()` verändern.

Es gibt sicherlich die ein oder andere Frage bezüglich des von den Unix-Entwicklern getroffenen Designs. Die eingehende Behandlung dieser Fragen würde aber den Rahmen dieser Lehrveranstalung bei weitem sprengen. Daher hier folgende Anmerkungen: Bei Eurem aktuellen Wissenstand müsst Ihr einfach akzeptieren, dass das Kernel-Design so ist, wie es ist. Es ist sogar sehr effizient und bietet hervorragende Möglichkeiten. Bei Interesse sei *fortgeschrittenen* Programmierern empfohlen, das Buch von Maurice Bach [1] zu lesen. Darüber hinaus hat jeder die Möglichkeit uns vor bzw. nach der Vorlesung oder während der Übung anzusprechen.

Kapitel 67

Dateien: Zusammenfassung

„Das war ja nun doch einiges an Hintergrundwissen. Was waren denn nun eigentlich die wichtigsten Punkte aus Programmierersicht?“ Ja, das war einiges. Dann halten wir mal folgende Dinge aus Programmierersicht fest:

1. Die Funktionen aus dem Paket `stdio.h` stellen eine komfortable Abstraktionsschicht der schlichten System Calls zur Verfügung.
2. Die Funktionen aus dem Paket `stdio.h` erkennen wir in der Regel daran, dass ihre Namen mit einem „`f`“ enden.
3. Viele der Funktionen aus dem Paket `stdio.h` benötigen einen File Pointer. Diese Funktionen erkennen wir daran, dass ihre Namen mit einem „`f`“ beginnen.
4. Wir können sowohl die Funktionen aus dem Paket `stdio.h` als auch die System Calls gleichzeitig verwenden. Aber wir tun gut daran, uns für die einen oder anderen zu entscheiden, da es sonst nur Komplikationen gibt.
5. Der Zusammenhang zwischen File Pointer `fp` und File Descriptor `fd` ist wie folgt gegeben: Wenn mittels eines Aufrufs der Funktion `fopen()` eine Datei geöffnet wird, wird zuerst der System Call `open()` aufgerufen, der einen File Descriptor zurückgibt. Dieser File Descriptor wird in die zuständige `FILE`-Struktur eingetragen, sodass wir `fileno( fp ) == fd` haben. Der File Pointer `fp` wird schließlich als Ergebnis des Aufrufs von `fopen()` zurückgegeben.
6. Die Funktionen aus dem Paket `stdio.h` rufen die System Calls eigenständig auf, sodass wir uns darum gar nicht kümmern müssen.

Teil VII

Professional C: dynamische Datenstrukturen

Kapitel 68

Inhalte dieses Skriptteils

Inzwischen haben wir fast alles an Sprachelementen durchgesprochen. Nur sind die Datenstrukturen noch ein wenig statisch: Wir müssen immer vor dem Übersetzen wissen, was wir wollen und wie viele Daten wir verarbeiten wollen; noch können wir uns nicht an die dynamischen Anforderungen der Nutzer anpassen.

In diesem Skriptteil lernen wir genau das! Man nennt sie dynamische Datenstrukturen. Darunter versteht man Dinge wie (lineare) einfach verkettete Listen, Bäume und Hash-Tabellen. Da die Konzepte der dynamischen Datenstrukturen nicht einfach sind, bereiten wir diese in den nächsten drei Kapiteln vor. Im Folgenden werden wir der Reihe nach folgende Themen behandeln:

Kapitel	Inhalt
69	Arbeitsspeicher auf Anforderung
70	Exkurs: dynamisches Anpassen von Datenstrukturen
71	Exkurs: Indirekt sortierte Arrays
72	Einfach verkettete Listen
73	Systematik von Listen
74	Der Stack
75	Sortierte Listen
76	Bäume
77	Hash-Tabellen

Auch in diesem Abschnitt gilt das schon öfter gesagte: Don't panic! Einfach mal ausprobieren und vor allem die Übungsaufgaben bearbeiten. Und bei Problemen helfen wir alle gerne weiter.

So, let's do it and have some (programming) fun!

Kapitel 69

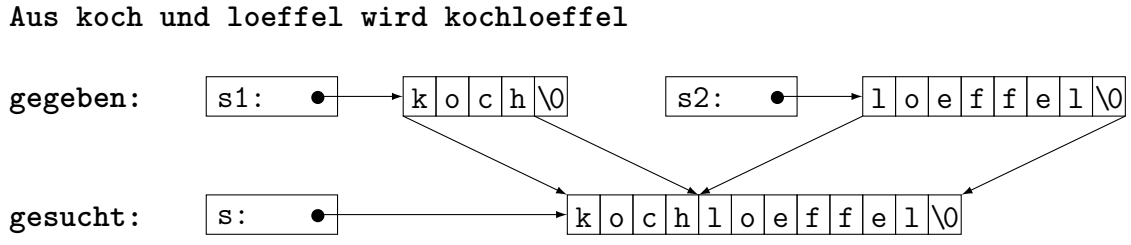
Arbeitsspeicher auf Anforderung

Ausgangspunkt dieses Kapitels ist folgende Problemstellung: Nehmen wir mal an, wir haben zwei mit einem Null-Byte abgeschlossene Zeichenketten `s1` und `s2`, die wir zu *einer* Zeichenkette zusammenfügen wollen. Beispiel: wir haben `koch` und `loeffel` und wollen daraus `kochloeffel` bauen. Um den `kochloeffel` im Arbeitsspeicher abzulegen brauchen wir auf jeden Fall zwölf Bytes. In diesem Kapitel besprechen wir, wie wir mittels der Standardfunktion `malloc()` so ein Speichersegment zur Laufzeit erhalten können. Bei Verwendung der Funktion `malloc()` sowie ihrer „Schwester“ `free()` sollten wir die Datei `stdlib.h` mittels `#include <stdlib.h>` einbinden.

69.1 Motivationsbeispiel: dynamische Zeichenketten

In diesem Abschnitt präsentieren wir ein einfaches Beispiel, um einen ersten Eindruck von der Verwendung von `malloc()` zu geben. Dabei lassen wir eine Reihe von Details unberücksichtigt, die wir erst in den folgenden Abschnitten klären werden. Wie gesagt, wir präsentieren hier nur ein kleines Beispiel, um einen ersten Eindruck zu vermitteln.

Aufgabe: Implementiere eine Funktion `char *concat( char *s1, char *s2 )` die zwei Zeichenketten übergeben bekommt, diese zu einer Zeichenkette zusammenfügt und einen Zeiger auf das Resultat zurückgibt. Diese Aufgabenstellung kann man wie folgt skizzieren:



Randbedingung: Wir haben folgende Randbedingungen zu beachten:

1. Zur Übersetzungszeit sind uns die Längen der beiden Zeichenketten `s1` und `s2` nicht bekannt; die Funktion `concat()` soll mit beliebigen Zeichenketten korrekt arbeiten.
2. Das Resultat soll auch nach dem Funktionsaufruf verfügbar sein.
3. Wir wissen zur Übersetzungszeit nicht, wie oft und mit welchen Zeichenketten die Funktion `concat()` aufgerufen wird. Daher ist die Definition eines Arrays fester Größe zum Ablegen der Ergebnisse ungeeignet.

Der traditionelle aber falsche Lösungsweg: Würden wir „blind“ drauflos programmieren, käme vielleicht folgende Lösung heraus:

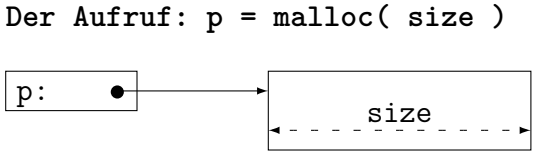
```
1 char *concat( char *s1, char *s2 )
2       {
3           char s[ 100 ];
4           strcpy( s, s1 );
5           strcpy( s + strlen( s1 ), s2 );
6           return s;
7       }
```

Das Kopieren der Zeichenketten `s1` und `s2` in den Zeilen 4 und 5 in das Ergebnis-Array ist korrekt. Aber aus folgenden Gründen erfüllt diese Lösung nicht die Aufgabenstellung:

1. Das Array `s` ist nach Ende der Funktion `concat()` wieder verschwunden, sodass wir es außerhalb der Funktion nicht verwenden können.
2. Auch ein `static char s[ 100 ]` würde nicht helfen. da ein erneuter Aufruf von `concat()` den Inhalt des „Ergebnisarray“ überschreiben würde.
3. Eine „konstante“ Definition eines Arrays `s` – ob nun mit oder ohne `static` ist ohnehin ungeeignet, da es bei längeren Zeichenketten `s1` und sources2 zu einem Speicherüberlauf käme.

Wir benötigen also einen anderen Lösungsweg.

Der dynamische Lösungsweg mittels `malloc()`: Der richtige Lösungsweg besteht in der Funktion `malloc()` mittels derer wir uns dynamisch zur Laufzeit des Programms weiteren Arbeitsspeicher organisieren können. Unten stehende Abbildung illustriert die Anweisung `p = malloc( size )`:



Die Funktion `malloc()` liefert also einen Zeiger auf den Anfang eines neuen Speichersegments, das `size` Bytes groß ist. Da dieses Speichersegment auf dem heap angelegt wird,

bleibt es bis zum Programmende oder einem entsprechenden Aufruf der Funktion `free()` im Arbeitsspeicher bestehen. Mit anderen Worten: Die Lebensdauer der auf dem Heap angelegten Segmente ist nicht an die Lebensdauer von Funktionen gebunden. Mit diesen Informationen können wir zu folgender Lösung kommen:

```
 1 #include <stdio.h>                                  // for printf()
 2 #include <stdlib.h>                                 // for malloc()
 3 #include <string.h>                                 // for strlen()
 4
 5 char *concat( char *s1, char *s2 )
 6      {
 7         char *p = malloc( strlen( s1 ) + strlen( s2 ) + 1 );
 8         if ( p != 0 )
 9            strcpy( strcpy( p, s1 ) + strlen( s1 ), s2 );
10         return p;
11      }
12
13 int main( int argc, char **argv )
14     {
15        char *p = concat( "koch", "loeffel" );
16        if ( p )
17           printf( "concat= '%s'\n", p );
18        else printf( "sorry, no space left in memory\n" );
19     }
```

Der entscheidende Funktionsaufruf ist in Zeile 7 zu sehen. Im Erfolgsfalle, d.h. der Zeigerwert ist ungleich `0`, werden wie im vorherigen Versuch die beiden Zeichenketten in das soeben erhaltene Speichersegment geschrieben. Da `strcpy` das erste Argument zurückgibt, haben wir die Kopieranweisungen zu Zeile 9 zusammengefasst. Die Funktion `concat()` gibt in Zeile 10 den Rückgabewert von `malloc()` mittels des Zeigers `p` an die aufrufenden Stelle zurück, sodass unser Hauptprogramm im Erfolgsfalle den Inhalt dieses Speichersegments ausgeben kann. Bei uns erscheint dann tatsächlich `concat= 'kochloeffel'`.

69.2 Verwendung

Die Verwendung von `malloc()` und `free()` läßt sich wie folgt darstellen:

C-Syntax	**Abstrakte Programmierung**
`ip = malloc(2*sizeof(int));`	`Alloziiere Speicher für 2 Int`
`cp = malloc( 20 );`	`Alloziiere Speicher für 20 Zeichen`
`free( cp );`	`Gib Speicherplatz wieder frei`

Hinweise: Bei Verwendung von `malloc()` und `free()` sind folgende Hinweise zu beachten:

1. Mit `malloc()` kann ein „beliebig“ großes Speichersegment beantragt werden.

2. Beim Aufruf von `malloc( size )` wird die Größe des Speichersegments in Vielfachen von Bytes angegeben.

3. Sofern das beantragte Speichersegment verfügbar ist, wird es dem laufenden Programm (Prozess) zugeordnet. Die Funktion `malloc()` gibt die Anfangsadresse dieses Speichersegments zurück. Beispiel: `p = malloc( some_bytes );`

4. Da `void *malloc( int )` einen Zeiger auf ein `void` zurückgibt, kann diese Adresse jedem beliebigen *Zeigertyp* ohne weiterem Cast zugewiesen werden; aber ein expliziter Cast schadet nie. Beispiel: `ip = (int *) malloc( 7 * sizeof( int ));`

5. Falls das beantragte Speichersegment nicht verfügbar ist, liefert `malloc()` einen Null-Zeiger zurück. Damit das anschließende Zugreifen auf das Segment nicht zu einem Programmabsturz führt, sollten alle von `malloc()` zurückgegebenen Zeigerwerte immer überprüft werden (siehe vorheriges Beispielprogramm).

   Bei heutigen PCs ist es selten, dass kein Speicherplatz mehr verfügbar ist. In der Regel sind derartige Probleme auf eine falsch berechnete Größe zurückzuführen. Daher sollte man bei einer Fehlermeldung auch diesen Wert ausgeben.

6. Ein mittels `malloc()` erhaltener Speicherplatz kann mittels `free()` wieder zurückgegeben werden. Beispiel: `p = malloc( some_bytes ); free( p );` Es ist *unbedingt* darauf zu achten, dass ein zurückzugebender Speicherbereich *vorher* auch mittels `malloc()` alloziiert wurde. `malloc()` und `free()` führen keinerlei Überprüfungen durch, sodass ein falscher Zeiger bei `free( p )` früher oder später zu einem Programmabsturz führt. Das Problem liegt vor allem im „später“, da das Finden derartiger Fehler *sehr* mühsam ist. Daher sollte man hier *sehr* sorgfältig programmieren.

7. Die Funktion `free( p )` kann mit einem Null-Zeiger aufgerufen werden, da dies ohne Effekt und somit korrekt ist.

69.3 Interne Repräsentation

Abgrenzung zu Stack Frames: Wir haben bereits mehrfach besprochen, wie jeder Funktionsaufruf den Stack und damit den belegten Arbeitsspeicher durch das Anlegen eines Stack Frames vergrößert. Bei Stack Frames gelten aber die folgenden beiden Randbedingungen: Erstens müssen die Variablen bereits bei der Programmerstellung *deklariert* (also bekannt) sein, denn sonst erscheinen sie nicht im Stack Frame. Zweitens werden am Ende eines Funktionsaufrufs die Stack Frames und damit ihre Variablen wieder vom Stack entfernt; damit sind auch die Variablen weg!

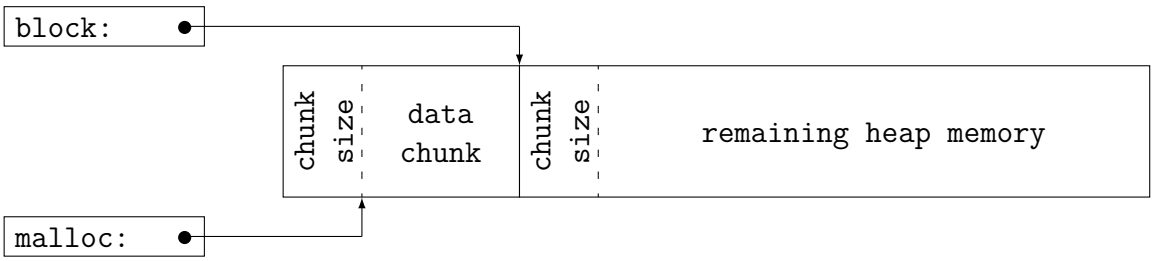
Das Heap-Segment: Über den Stack hinaus bieten moderne Betriebssysteme wie Linux mindestens zwei Möglichkeiten, das Heap-Segment (siehe auch Kapitel 40) zu vergrößern. Die entsprechenden Betriebssystemfunktionen heißen `sbrk()` und `mmap()`. Da derartige

Betriebssystemaufrufe immer sehr teuer im Sinne von Rechenzeit sind, werden diese durch `malloc()` und `free()` verborgen. Das Ergebnis sind zwei Funktionen, die für den Programmierer leicht zu verwenden sind und die die Zusammenarbeit mit dem Betriebssystem stark vereinfachen. Diese beiden Funktionen arbeiten im wesentlichen wie folgt[1]:

1. Sie besorgen sich vom Betriebssystem mittels des System Calls `sbrk()` einen größeren Speicherblock, beispielsweise 32 KB.

2. `malloc()` löst auf Anforderung hin den benötigten Speicherbereich aus diesem Block heraus. Dies läßt sich etwas vereinfacht wie folgt darstellen:

   Der Aufruf von `malloc()`

   Die Grafik veranschaulicht folgendes: `malloc( size )` reserviert nicht nur einen Speicherblock der Größe `size`-Bytes sondern *zusätzlich* noch ein `int` (`sizeof(int)` Bytes). In diesen zusätzlichen `int`-Bereich schreibt `malloc()` die Größe des reserviert Speicherblocks, sodass beide Funktionen diese Größe immer wieder ermitteln können. Der Rückgabewert von `malloc()` ist dann nicht der Anfang dieses Speicherblocks sondern ist genau um ein `int` verschoben, sodass diese kritische Angabe nicht überschrieben wird.

3. Die Funktion `free()` arbeitet genau entgegengesetzt: Sie nimmt einen Block, geht um die Größe einer `int`-Zahl nach unten (in Richtung kleinerer Adresse), findet dort wieder die Größe des zurückgegebenen Speicherblocks und versucht, diesen in den großen, gemeinsam verwalteten Block einzufügen.

4. Sollte der Heap-Speicher leer oder zu klein sein, besorgt sich `malloc()` einen weiteren großen Speicherblock vom Betriebssystem. Auf meinem Rechner sind dies beispielsweise Vielfache von 2 KB.

Problemfall: falsches `free()`: Aufgrund obiger Erklärungen, insbesondere der beiden Punkte 2 und 3, sollte klar geworden sein, dass das Zurückgeben eines falschen, also gar nicht durch `malloc()` alloziierten Speicherblocks über kurz oder lang zu einem Problem führt: der Zeiger zeigt an eine falsche Stelle, die sich hinter dem Zeiger befindliche Größenangabe ist demnach vermutlich falsch, `free()` nimmt aufgrund der falschen Größenangabe unter Umständen an, dass der Speicherblock viel größer als in Wirklichkeit ist, dadurch

[1] Auch diesmal ist die Beschreibung überraschenderweise ein wenig vereinfacht, aber doch sehr dicht an der wirklichen Implementierung.

werden Speicherbereiche als frei markiert, die tatsächlich noch belegt sind, daraus resultieren weitere Inkonsistenzen, die letztlich irgendwann zu einem Programmabsturz führen. Unglücklicherweise treten derartige Programmabstürze in der Regel viel später auf als der ursächliche Aufruf von `free()`. Daher: viel Vergnügen bei der Fehlersuche.

Lebensdauer: Die Lebensdauer des auf dem Heap alloziierten Speicherbereichs ist *nicht* an die Lebensdauer irgendeiner Funktion gekoppelt. *„Bitte was?“* Genau, der mittels `malloc()` alloziierte Speicher existiert bis er entweder mittels eines entsprechenden Aufrufs von `free()` explizit freigegeben oder das Programm beendet wird. Man muss sich aber die Adresse dieses Speicherblocks in einer Zeigervariablen gut merken, denn sonst findet man ihn nicht mehr wieder. Hierzu folgendes Beispiel:

```
 1 #include <stdio.h>                          // for printf()
 2 #include <stdlib.h>                         // for malloc()
 3
 4 int *my_new_int( int i )
 5     {
 6        int *p = malloc( sizeof( int ) );
 7        if ( p )
 8           *p = i;                           // set value;
 9        return p;
10     }
11
12 int main( int argc, char **argv )
13     {
14        my_new_int( 4711 );
15        // damned, 4711 is stored, but the address is lost!
16     }
```

Korrekt und notwendig wäre gewesen, sich die Adresse in einer Zeigervariablen zu merken: `int *p = my_new_int( 4711 )`. Anschließend könnte man sich den Wert ausgeben lassen: `printf( "Wert der neuen Vairablen: %d\n", *p )` Um es noch einmal klar und deutlich zu sagen: hat man vergessen, sich die Adresse zu merken, ist sie für immer verloren; der Speicherplatz ist aber dennoch als belegt markiert.

Abgrenzung zu regulären Variablen: Bisher hatten wir eigentlich nur reguläre Variablen. Diese hatten einen Namen, für die der Compiler eine Adresse (auf dem Stack) festgelegt hat. Alle Zugriffe auf diese Variablen, egal ob nun lesend oder schreibend, haben wir über ihren Namen bewerkstelligt; der Compiler hat daraus Zugriffe auf den Arbeitsspeicher mit den entsprechenden Adressen gemacht. Bei den dynamisch generierten Variablen ist dies grundsätzlich anders: Diese Variablen haben zwar eine Adresse, die wir uns auch noch selbst merken müssen, aber keinen Namen, weshalb man sie auch als *anonyme* Variablen bezeichnet. Da diese Variablen keinen Namen haben, kann der Compiler keine Adressen[2]

[2]Dies geht prinzipiell nicht, da sich die Adressen erst zur Laufzeit ergeben.

bestimmen, sodass wir diese Arbeit durch eine Zeigervariable und anschließendem Dereferenzieren selbst erledigen müssen. *„Und das soll cool oder nützlich sein?“* Ja! Am Anfang mutet es etwas umständlich an, aber auf Dauer gewöhnt man sich daran. Und vor allem, wir können nahezu beliebig viele neue anonyme Variablen zur Laufzeit generieren. Doch das machen wir erst ab Kapitel 72.

69.4 Beispiele

In diesem Abschnitt präsentieren wir ein paar einfache Beispiele, damit ihr Euch an das dynamische Organisieren von Speicherplatz gewöhnen könnt.

Allokation von 26 Zeichen: Im folgenden Programmbeispiel wird ein Array mit 26 Zeichen alloziiert, deren Elemente der Reihe nach mit den Buchstaben von `'a'` bis `'z'` belegt werden. Anschließend werden diese Zeichen ausgegeben.

```
1  #include <stdio.h>
2  #include <stdlib.h>                                  // for malloc()
3
4  int main( int argc, char **argv )
5      {
6          char *cp = malloc( 26 );                    // 26 characters
7          int i;
8          if ( cp )
9             for( i = 0; i < 26; i++ )
10                cp[ i ] = 'a' + i;                   // initialization
11         if ( cp )
12            for( i = 0; i < 26; i++ )
13               printf( "cp[ %2d ] = '%c'\n", i, cp[ i ] );
14     }
```

Allokation eines eigenen `struct`: Im folgenden Beispiel (oben, nächste Seite) wird in Zeile 12 eine selbst definierte Struktur (Zeilen 4 bis 8) dynamisch organisiert. Die Ausgabe in Zeile 16 ist erwartungsgemäß `i= 4711 c= x d= 3.141`.

69.5 Zusammenfassung

Fassen wir noch einmal zusammen: Mit `malloc()` können wir uns zur Laufzeit beliebig viel Speicherplatz besorgen. Wir müssen nur dessen Größe wissen, wobei uns die Compiler-Funktion `sizeof()` hilfreich zur Seite steht. Der von `malloc()` organisierte Speicherplatz wird nicht auf dem Stack sondern auf dem Heap angelegt, weshalb er *nicht* an die Lebensdauer der umgebenden Funktion gebunden ist; bezüglich der Lebensdauer verhält sich der Heap wie das Data-Segment, das die Lebensdauer des Programms hat und die globalen

```
1  #include <stdio.h>
2  #include <stdlib.h>              // fuer malloc()
3
4  typedef struct {
5                  int i;
6                  char c;
7                  double d;
8          } my_struct;
9
10 int main( int argc, char **argv )
11     {
12        my_struct *p = malloc( sizeof( my_struct ) );
13        if ( p )
14        {
15          p->i = 4711; p->c = 'x'; p->d = 3.141;
16          printf( "i= %d c= %c d= %5.3f\n", p->i, p->c, p->d );
17        }
18     }
```

Variablen beherbergt. Mittels `free()` können wir den zuvor organisierten Speicherplatz wieder freigeben, sofern wir ihn wirklich nicht mehr benötigen.

Der etwas unangenehme Part ist, dass uns der Compiler nicht mehr die Arbeit durch die Kombination `Name/Adresse` abnimmt; wir müssen uns selbst um diese Details kümmern. Diese eigene explizite Verwaltung der Adressen scheint momentan eigentlich nur umständlich: statt eine einfache Variable zu deklarieren benötigen wir jetzt einen Zeiger (grusel), einen Funktionsaufruf (`malloc()`) und eine Fehlerüberprüfung (`if (p == 0)`). Der einzige Vorteil besteht momentan für uns darin, dass wir uns um die Variablen super flexibel zur Laufzeit kümmern können. In den nächsten Kapiteln werden wir sehen, wie und wo wir das Konzept der dynamischen Datenstrukturen gewinnbringend einsetzen können.

Kapitel 70

Exkurs: dynamisches Anpassen von Datenstrukturen

Im Grunde genommen könnten wir mit dem, was wir über `malloc()` im lezten Kapitel gelernt haben, schon fast alles machen. Beispielsweise könnten wir die Größe eines Arrays „zur Laufzeit dynamisch anpassen.“ Das wäre nicht besonders elegant und auch nicht besonders effizient, aber es ginge und wir hätten damit auch eine Basis für weitere dynamische Datentruktren wie Listen, Bäume und Hashtabellen; in Ermangelung anderer Programmiertechniken wurden diese früher auch mittels Arrays realisiert. Auch wenn wir die eben abgesprochenen Techniken wie Listen, Bäume und Hashtabellen in den Kapiteln 72 bis 77 auf anderem Wege etablieren, verfolgen wir hier in diesem Kapitel dennoch den oben angesprochenen Weg der dynamischen Arrays. Dies machen wir aus zwei Gründen: erstens bekommen wir auf einfachem Weg etwas mehr Übung und zweitens können wir die Vor- und Nachteile der einzelnen Techniken nachher etwas besser voneinander abgrenzen.

70.1 Dynamisch wachsende Arrays

Problemstellung: In vielen Anwendungen stellt sich erst während der Verarbeitung heraus, wie viele Daten zu verarbeiten sind. Derartige Aufgabenstellungen können wir normalerweise mittels folgender Schleife bearbeiten:

```
1 int i;
2 do {
3       printf( "Bitte positive Zahl eingeben: " );
4       scanf( "%d", & i );
5       // processing comes here
6     } while( i > 0 );
```

Schwierig wird diese Aufgabenstellung, wenn wir die Zahlen nicht nur verarbeiten sondern aus irgendeinem Grunde auch noch abspeichern müssen. Klar, ein Array wäre das beste.

Doch kennen wir die korrekte Größe des benötigten Arrays erst *nach* Beendigung der Schleife, also zu spät. Ein Array fester Größe können wir auch nicht nehmen, da wir keinen Anhaltspunkt über eine eventuelle Maximalgröße haben. Wir brauchen also ein Array, dessen Größe wir von Zeit zu Zeit zur Laufzeit für jede neue Zahl anpassen können.

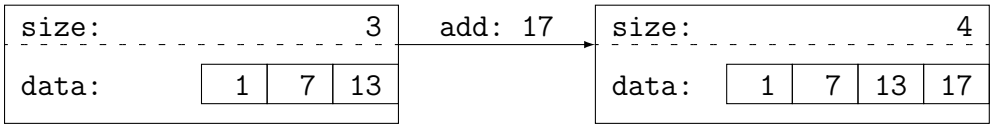
Aufgabenstellung: Implementiere einen abstrakten Datentyp „Dynamisches Array“, dessen Größe zur Laufzeit dynamisch wachsen kann. Angewendet auf unser obiges Beispiel bedeutet dies: Vergrößer das Array um ein Element, wann immer eine neue Zahl kommt.

Entwurf: Der abstrakte Datentyp „Dynamisches Array“ besteht aus einer Struktur und einer Funktion `grow_da()`. Die Struktur hat zwei Felder, eines für die aktuelle Größe des Arrays und eines für das eigentliche Daten-Array. Die Funktion `grow_da()` ist in der Lage, das Daten-Array um ein Element zu vergrößern.

Implementierung: Aus den obigen Bemerkungen können die folgenden fünf Schritte direkt abgeleitet werden:

1. Alloziieren eines *neuen* Arrays, das ein Element größer ist als das bestehende.
2. Kopieren der vorhandenen Daten in das neue Array.
3. Speichern der neuen Zahl im neu hinzugekommenen Element.
4. Anpassen der Größe des neuen Arrays.
5. Freigeben des alten Array.

Das folgende Bild zeigt, wie zu den drei Zahlen `1`, `7` und `13` die Zahl `17` hinzukommt:



Kodierung: Das resultierende C-Programm ist auf der nächsten Seite abgebildet. Da das Programm relativ klar sein sollte, folgen hier nur ein paar Anmerkungen:

1. In den Zeilen 4 bis 7 wird ein strukutrierter Datentyp definiert, der aus einem Zeiger auf ein Array `da_data` sowie der Angabe der aktuellen Größe `da_size` besteht.
2. Die Funktion `grow_da()` (Zeilen 9 bis 21) ist für die Vergrößerung des Arrays zuständig. Ihre Kodierung setzt die obigen fünf Schritte direkt um.
3. In den Zeilen 27 und 30 des Hauptprogramms wird jeweils eine neue Zahl `i` eingelesen und wird nur dann in das Array eingefügt, wenn die Bedingung `i > 0` erfüllt ist. Gemäß Kapitel 43 wird der zweite Teil der logischen „und“-Bedingung in Zeile 30 nur dann ausgewertet, wenn der erste Teil bereits logisch wahr ergab.
4. Die gewählte Formulierung der Schleifenbedingung in Zeile 30 hat auch den Vorteil, dass die Schleife dann abbricht, wenn kein neuer Speicher mehr verfügbar ist.
5. Bei Eingabe der drei Zahlen `4711`, `4712` und `4713` werden diese wieder ausgegeben.

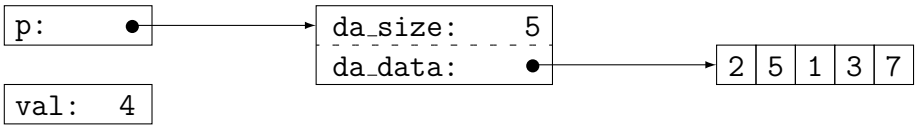
```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct {
5                  int da_size;             // current array size
6                  int *da_data;           // the actual array
7          } DYNAMIC_ARRAY, *DA_PTR;      // type struct & pointer
8
9  int grow_da( DA_PTR p, int val )         // add a new value
10     {
11        int i, *new = malloc( sizeof(int) * (p->da_size + 1));
12        if ( new )                      // was malloc successful?
13        {
14           for( i = 0; i < p->da_size; i++ )
15              new[ i ] = p->da_data[ i ]; // save/copy old data
16           new[ p->da_size++ ] = val;  // add val and increment
17           free( p->da_data );         // free old array if any
18           p->da_data = new;              // save the new array
19        }
20        return new != 0;                  // all ok
21     }
22
23 int main( int argc, char **argv )
24     {
25        int i;
26        DYNAMIC_ARRAY da = {0, 0};       // initialize properly
27        do {
28             printf( "Bitte positive Zahl eingeben: " );
29             scanf( "%d", & i );
30           } while( i > 0 && grow_da( & da, i ));
31        for( i = 0; i < da.da_size; i++ )
32           printf( "%d\n", da.da_data[ i ] );
33     }
```

Grafische Darstellung: Auf der folgenden Seite wird die Arbeitsweise der Funktion `grow_da()` grafisch dargestellt. Die gestrichelt eingezeichneten Pfeile repräsentieren keine Zeiger, sondern deuten darauf hin, welche Änderung in welcher Programmzeile passiert.

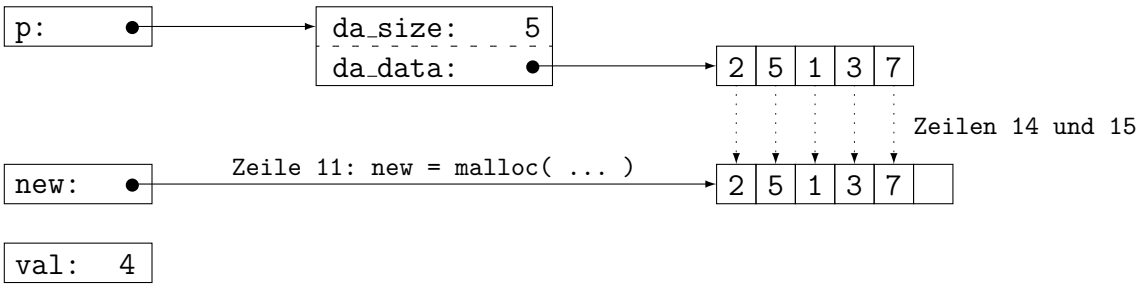
70.2 Diskussion

Das Schöne an diesem Beispielprogramm ist, dass es wie gewünscht funktioniert und es ein schönes Beispiel für die Verwendung des Heaps ist. Ferner ist auch der Quelltext einigermaßen kompakt und übersichtlich ohne großartige Sonderfälle oder ähnliche Dinge. Allerdings

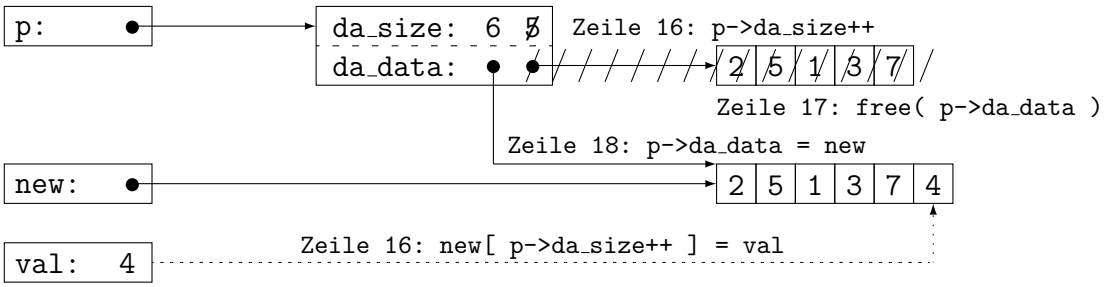
Funktionsaufruf: grow_da (& da , i)



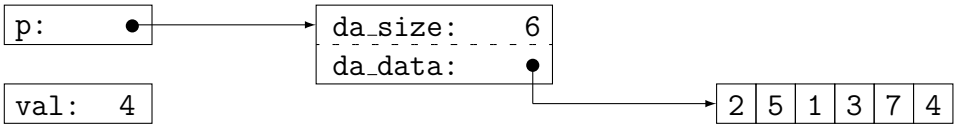
Funktionsabarbeitung: Zeilen 11 bis 15



Funktionsabarbeitung: Zeilen 16 bis 18



Funktionsende



hat dieser Ansatz auch einen gravierenden Nachteil: Beim Hinzufügen eines einzelnen Elementes muss jeweils das gesamte, bereits existierende Array kopiert werden. Bei n Zahlen sind dies $n \times (n-1)/2$ Kopiervorgänge, was bei größeren Zahlenmengen schnell einmal die notwendige Rechenzeit erhöht. Die Ursache für die Notwendigkeit, jedesmal das gesamte Array kopieren zu müssen, liegt darin, dass die Elemente eines Arrays bündig aneinandergereiht im Arbeitsspeicher liegen müssen. Um dieses Problem zu eliminieren, müssten wir zwei Dinge tun können: Erstens, müssten wir das Array in seine einzelnen Elemente zerschneiden können und zweitens müssten wir immer wissen, wo sich das nächste Element des Arrays im Arbeitsspeicher befindet. Dies ist gar nicht so schwer und besprechen wir im nächsten Kapitel.

Kapitel 71

Exkurs: Indirekt sortierte Arrays

Wenn wir ein Array sortieren, beispielsweise aufsteigend, dann ist das nächste Element größer oder gleich dem vorherigen. Mathematisch könnten wir dies so formulieren: Für alle Elemente x_i gilt: $x_i \leq x_{i+1}$. Wenn wir nun ein neues Element einfügen wollen, dass aufgrund seines Wertes irgendwo in die Mitte gehört, müssen wir ungefähr die Hälfte der Elemente um eine Position nach hinten verschieben. Bei großen Arrays kostet dies Zeit. Schön wäre es doch, wenn wir das neue Element einfach hinten anfügen würden und das Array trotzdem sortiert bleibt. *„Wie soll das denn gehen? Das ist doch totaler Quatsch... Das kann doch gar nicht gehen, wenn die richtige Position irgendwo in der Mitte ist.“* Nun, immer schön langsam mit den wilden Pferden. Natürlich geht so etwas nicht gratis. Aber wir können ja mal etwas probieren.

71.1 Expliziter Nachfolger: ein erweitertes Konzept

Bei sortierten Arrays gehen wir implizit immer davon aus, dass die Position eines Elementes mit dem Index innerhalb des Arrays identisch ist. Das ist natürlich oft auch gut so. Aber, dies muss nicht immer so sein. Wir könnten die Elemente auch so gestalten, dass sie neben den Daten auch die Position des nächsten Elementes innerhalb des Arrays beinhalten.

„Hä?“ Ok, die Fragezeichen sind riesig. Also, ein Beispiel. Doch zuvor noch eine kleine Bemerkung: Gültige Array-Indizes sind immer größer oder gleich null. Keines der Elemente hat einen negativen Index, zumindest so lange wir uns innerhalb des Arrays bewegen. So, was macht das Programm auf der folgenden Seite?

Es druckt die Zahlen: `liste: 2 9 13 18 22 70 78 99`. *„Geil! Zugegeben, ein wenig merkwürdig, aber dennoch einen Gedanken wert, denn die Zahlen sind aufsteigend sortiert“* Genau. Dadurch, dass die Schleife in Zeile 16/17 das Array nicht mehr Element für Element durchgeht, sondern den nächsten Index `next` aus dem Element selbst bezieht (dritter Ausdruck in Zeile 17), werden die Werte `value` aufsteigend sortiert ausgegeben.

```
1  #include <stdio.h>
2
3  typedef struct {
4                  int value;              // der Wert
5                  int next;               // der naechste Index
6          } ELEMENT, *EP;
7
8  int main( int argc, char **argv )
9      {
10         ELEMENT liste[] = {
11                  { 22, 1 }, { 70, 5 }, { 18,  0 }, {  9,  7 },
12                  {  2, 3 }, { 78, 6 }, { 99, -1 }, { 13,  2 }
13                 };
14         int first = 4, i;
15         printf( "liste:" );
16         for( i = first; i != -1; i = liste[ i ].next )
17            printf( " %d", liste[ i ].value );
18         printf( "\n" );
19     }
```

Bevor jetzt irgendjemand etwas falsch versteht: Wir sagen nicht, dass man so programmieren soll. Obiges kleines Beispiel ist mehr als konzeptuelles Gedankenspiel zu betrachten. Denn mit den Konzepten, die wir in diesem und im vorherigen Kapitel besprochen haben, werden wir im nächsten Kapitel das Konzept der einfach verketteten Listen entwickeln. Im nächsten Abschnitt werden wir unser Beispiel noch ein wenig weiter entwickeln, um im nächsten Kapitel einige Aspekte noch weiter zu vereinfachen.

71.2 Zwei Erweiterungen für reale Anwendungen

Problempunkte: Obiges Beispiel hatte lediglich einen einzigen Nutzwert pro Element. In einer realen Anwendung kann dies natürlich wesentlich komplexer sein. In solchen Fällen ist es sinnvoll, die eigentlichen Daten von den „Verwaltungsdaten“ (wie `next`) sauber zu trennen. Warum soll eine Funktion, die den Datensatz ausgibt, etwas vom nächsten Index `next` wissen? Muss sie nicht, sollte sie auch nicht! Ferner ist es recht unschön, dass sich die eigentlichen Daten im Array `liste` befinden, der erste Index aber außerhalb des Arrays in der Variablen `first` abgelegt ist. Diese beiden Punkte werden wir im Folgenden bearbeiten.

Trennung von Nutzdaten und Verwaltungsdaten: Diese Trennung können wir einfach erreichen, in dem wir für die Daten eine eigene Struktur definieren. Zur Illustration reichern wir den Datensatz um eine weitere Komponente an, die vom Datentyp `char` ist. Die Typdefinitionen könnten wie folgt aussehen:

```
3  typedef struct {
```

```
 4                   int  ival;     // der eine Wert
 5                   char cval;     // der andere Wert
 6           } DATA, *DP;           // Typ: Daten & Pointer
 7
 8  typedef struct {
 9                   int next;      // naechster Index (Verwaltung)
10                   DATA data;     // die Nutzdaten
11           } ELEMENT, *EP;        // Typ: Daten & Pointer
```

Index des ersten Elementes: Das Einfachste ist, den Index des ersten Elementes mit in das Array zu integrieren. Hierzu könnten wir einfach ein „Dummy“ Element als erstes Element allen Datenelementen voranstellen. Dann würden wir von diesem Element nicht die Daten sondern nur die Verwaltungsdaten (`next`) nutzen. Die konkrete Implementierung könnte wie folgt aussehen:

```
20           ELEMENT liste[] = {
21                     { 5, {} }, ...  // an Position 5 geht es los
```

Drucken der Nutzdaten: Zur weiteren Illustration implementieren wir gleich noch eine Funktion, die einen Zeiger auf die Nutzdaten erhält und die Werte entsprechend ausgibt:

```
13  int prt_data( DP dp )            // Funktion zum Drucken
14      {
15          printf( "\tival= %2d cval= '%c'\n",dp->ival,dp->cval);
16      }
```

Das vollständige Programm: Unter Zuhilfenahme aller oben besprochenen Erweiterungen sieht das vollständige Programm so aus, wie auf der folgenden Seite abgedruckt.

Schlussbemerkung: Auch wenn der hier beschriebene Weg aus heutiger Sicht etwas antiquiert aussieht, so dient er dennoch der einfacheren Einführung des Konzepts der einfach verketteten Listen. Ferner sollte man nicht vergessen, dass man früher in anderen Programmiersprachen genau so Listen und andere dynamische Datenstrukturen programmiert hat, da sie keine Möglichkeiten boten, dynamisch Speicherplatz zu alloziieren. Weiterhin kann man noch anmerken, dass man die Flexibilität noch weiter erhöht, wenn man das `struct` für die Nutzdaten nicht direkt in den Element-Datensatz integriert, sondern über einen Zeiger auf sie verweist. Und wenn man diesen Zeiger noch als `void *` deklariert, kann man alle Funktionen recht generisch auslegen. Aber das sprengt hier den Rahmen.

```
 1 #include <stdio.h>
 2
 3 typedef struct {
 4                 int  ival;     // der eine Wert
 5                 char cval;     // der andere Wert
 6         } DATA, *DP;           // Typ: Daten & Pointer
 7
 8 typedef struct {
 9                 int next;      // naechster Index (Verwaltung)
10                 DATA data;     // die Nutzdaten
11         } ELEMENT, *EP;        // Typ: Daten & Pointer
12
13 int prt_data( DP dp )          // Funktion zum Drucken
14     {
15         printf( "\tival= %2d cval= '%c'\n",dp->ival,dp->cval);
16     }
17
18 int main( int argc, char **argv )
19     {
20         ELEMENT liste[] = {
21                   { 5, {         }}, {  2, { 22, 'a' }},
22                   { 6, { 70, 'h' }}, {  1, { 18, 'L' }},
23                   { 8, {  9, 'z' }}, {  4, {  2, 'p' }},
24                   { 7, { 78, 'A' }}, { -1, { 99, 'o' }},
25                   { 3, { 13, 'Q' }}
26                 };
27         int i;
28         printf( "liste:\n" );
29         for( i = liste[ 0 ].next; i != -1; i = liste[i].next )
30            prt_data( & (liste[ i ].data) );
31     }
```

Ausgabe:

```
liste:
        ival=  2 cval= 'p'
        ival=  9 cval= 'z'
        ival= 13 cval= 'Q'
        ival= 18 cval= 'L'
        ival= 22 cval= 'a'
        ival= 70 cval= 'h'
        ival= 78 cval= 'A'
        ival= 99 cval= 'o'
```

Kapitel 72

Einfach verkettete Listen

In den beiden vorherigen Kapiteln kamen wir zu folgenden Erkenntnissen:

1. Mittels `malloc()` und Kopieren kann man Arrays zur Laufzeit vergrößern und damit dynamisch an einen erhöhten Speicherbedarf anpassen.
2. Das bei der dynamischen Erweiterung von Arrays notwendige Kopieren kann auf Dauer sehr viel Ressourcen (vor allem Rechenzeit) binden.
3. Arrays kann man auf verschiedene Weisen sortieren, es muss nicht notwendigerweise `x[ i ] <= x[i + 1]` gelten. Allerdings muss man den Nachfolger (das nächste Element) selbst explizit verwalten.
4. Arrays verlangen, dass ihre Elemente bündig aneinander im Arbeitsspeicher liegen.

In diesem Kapitel werden wir nun diese Ergebnisse zusammenführen und daraus das Konzept der einfach verketteten Listen entwickeln.

72.1 Arrays zerschneiden: ein erster Lösungsversuch

Das Hauptproblem in den beiden letzten Kapiteln war, dass die Elemente eines Arrays bündig aneinandergereiht im Arbeitsspeicher liegen müssen (Punkt 4 in obiger Liste). Also stellen wir uns doch einfach mal folgende Frage:

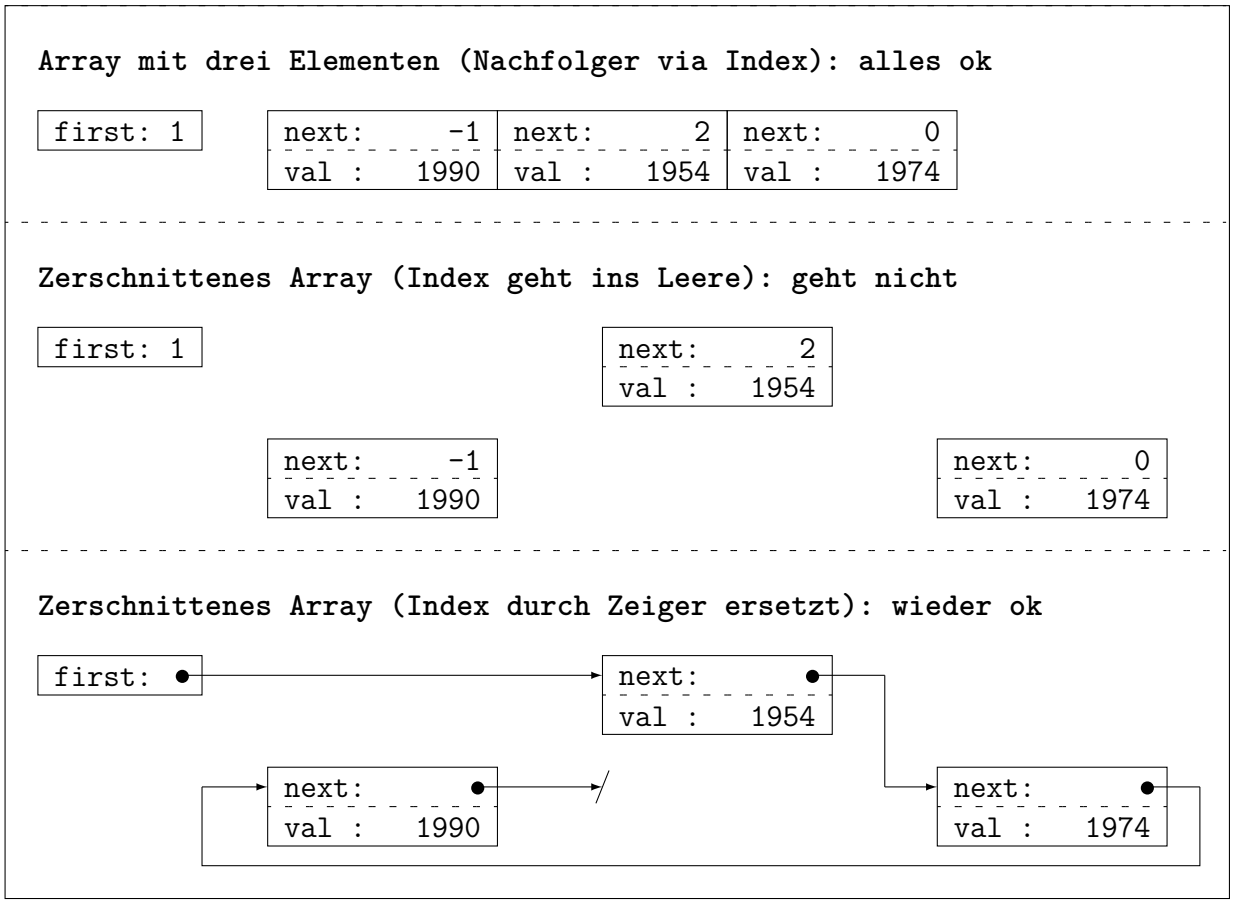
> Was müssen wir alles ändern, wenn wir ein Array in seine einzelnen Elemente zerschneiden und diese (beliebig) im Arbeitsspeicher verteilen?

Die Anwort ist verblüffend einfach: Nicht viel, wir müssen nur aus den Nachfolgeindizes echte Adressen machen. Das war's!

Bevor wir uns an die Umsetzung machen, erinnern wir uns nochmals kurz an ein paar Details bezüglich Arrays, Adressen und Adressberechnungen. Wer hier unsicher ist, lese am besten nochmals die Kapitel 45 und 46. Wir wissen folgendes:

1. Bei der Definition `some_type array[ size ]` haben die einzelnen Elemente den Typ `some_type`.
2. Ein Zeiger auf ein derartiges Element definiert man wie folgt: `some_type *ptr`.
3. Die Adresse `& array[ i ]` eines Elementes `i` ist nichts Anderes als `array + i`.
4. Der Zugriff `array[ i ] = x` auf ein Array-Element `i` ist identisch mit folgender Zeigerarithmetik: `*(array + i) = x`.

Die ganze Idee haben wir für Euch am Beispiel einfacher Zahlen ein wenig illustriert:



72.2 Schrittweise Umsetzung

Der Datentyp: Der erste wesentliche Punkt ist die Definition eines geeigneten Datentyps. Hierfür brauchen wir ein `struct`, da wir verschiedene Datentypen zusammenfassen wollen. Das Problem ist nun aber, dass man auf ein `struct` erst zugreifen kann, wenn man mit der Typdefinition fertig ist. Da dieses Problem aber von genereller Natur ist und da alle

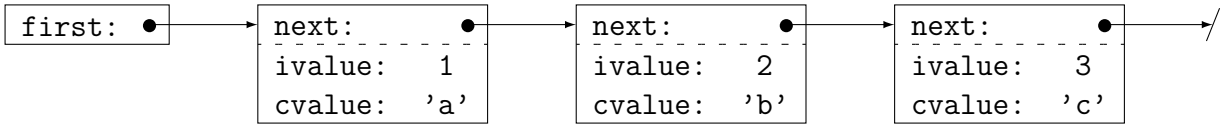
Zeiger in C gleich groß sind, kann man bereits innerhalb eines `structs` einen Zeiger auf selbigen Typ definieren. Eine derartige Typdefinition könnte wie folgt aussehen:

```
 9  typedef struct _element {        // struct heisst '_element'
10                  struct _element  // Adresse des naechsten
11                         *next;    // Elementes (Verwaltung)
12                  DATA data;       // die eigentlichen Nutzdaten
13          } ELEMENT, *EP;          // Typ: Daten & Pointer
```

Hierbei ist `DATA` wieder das `struct`, das wir bereits im vorherigen Kapitel definiert haben, was uns auch die Übernahme der Ausgabefunktion `prt_data()` erlaubt. Diese Typdefinition hat zwei kleine Besonderheiten. Erstens hat das `struct` einen vorläufigen Namen, der in diesem Fall `_element` lautet. Dadurch können wir in den Zeilen 10 und 11 einen Zeiger `next` definieren, der auf ein Element selbigen Datentyps zeigt.

Ausgabe der Liste: Wie im Beispiel des vorherigen Kapitels benötigen wir einen Anfangszeiger. Diesen werden wir im Hauptprogramm definieren und `first` nennen. Im Beispiel des vorherigen Kapitels haben wir das Ende der Liste mittels des Index' `-1` kenntlich gemacht. Dies war notwendig, da der Index `0` für jedes beliebige Array gültig ist. Aus Kapitel 45.8 wissen wir bereits, dass der Zeigerwert `0` angibt, dass es sich um keine gültige Adresse handelt und wir ihn somit als Endekennung einer Liste verwenden können. Den eigentlichen Aufbau der Liste werden wir noch ein bisschen hinausschieben; hier gehen wir jetzt davon aus, dass folgende Liste im Arbeitsspeicher vorliegt, die wir in `main()` sehr einfach wie folgt ausdrucken können:

Eine Liste mit drei Elementen




```
33          EP first = 0, p;
34          // listenaufbau: first --> | *-|-->| *-|-->| *-|-->/
35          //                         | 1 |   | 2 |   | 3 |
36          //                         |'a'|   |'b'|   |'c'|
37          printf( "Liste:\n" );
38          for( p = first; p != 0; p = p->next )
39             prt_data( & p->data );
```

Aufbau der Liste: Für den Aufbau der Liste benötigen wir eigentlich nur eine Funktion `mk_element()`, die mittels `malloc()` (siehe auch Kapitel 69) ein neues Element auf dem Heap erzeugt. Eine derartige Funktion könnte wie folgt aussehen:

```
20 EP mk_element( int i, char c, EP next )
21    {
22       EP p = malloc( sizeof( ELEMENT ) );
23       if ( p != 0 )              // malloc() war erfolgreich?
24       {
25          p->next = next;         // initialisierung
26          p->data.ival = i; p->data.cval = c;
27       }
28       return p;                  // Adresse des neuen Elementes
29    }
```

Diese Funktion ist recht einfach. Sie besitzt drei Parameter, die bei erfolgreichem Aufruf der Funktion `malloc()` der Initialisierung der einzelnen Komponenten des verschachtelten `struct`s dienen (Zeilen 25 und 26). Die Liste kann nun wie folgt aufgebaut werden:

```
33         EP first = 0;
34         first = mk_element( 3, 'c', first );
35         first = mk_element( 2, 'b', first );
36         first = mk_element( 1, 'a', first );
```

Man beachte, dass der alte Wert von `first` mittels `mk_element()` gerettet und anschließend mit der Adresse des neuen Elementes (dem Funktionswert) überschrieben wird.

Das vollständige Programm: Aus Platzgründen ist das vollständige Programm auf der nächsten Seite wiedergegeben.

72.3 Zusammenfassung

Ok, erst mal tief durchatmen und dann nochmals alles Revue passieren lassen. Im Vergleich zu den beiden vorherigen Kapiteln bestand die wesentliche Erweiterung darin, dass wir das „alte“ Array in seine Elemente zerschnitten und diese mittels der Funktion `malloc()` bedarfsweise alloziiert haben. Durch das „Auseinanderschneiden“ des Arrays konnten wir nicht mehr an den Indizes zum Verketten der einzelnen Elemente festhalten und mussten diese durch explizite Zeiger ersetzen. Das war's eigentlich schon. Glückwunsch zur ersten einfach verketteten Liste!

In den nächsten fünf Kapiteln werden wir uns einzelne Aspekte dieser dynamischen Listen etwas genauer ansehen. Im folgenden Kapitel werden wir erst einmal die gängigsten Listentypen kurz erläutern. Dieses Kapitel diente nur dazu, einmal einen ersten Eindruck von dynamischen Listen zu bekommen.

Das vollständige Programm:

```
 1 #include <stdio.h>              // fuer die Ein-/Ausgabe
 2 #include <stdlib.h>             // fuer malloc()
 3
 4 typedef struct {                // erst unsere alten Daten
 5                 int  ivalue;    // der eine Wert
 6                 char cvalue;    // der andere Wert
 7         } DATA, *DP;            // Typ: Daten & Pointer
 8
 9 typedef struct _element {       // struct heisst '_element'
10                 struct _element // Adresse des naechsten
11                         *next;  // Elementes (Verwaltung)
12                 DATA data;      // die eigentlichen Nutzdaten
13         } ELEMENT, *EP;         // Typ: Daten & Pointer
14
15 int prt_data( DP dp )           // Funktion zum Drucken
16     {
17         printf( "\ti= %2d c= '%c'\n", dp->ivalue,dp->cvalue );
18     }
19
20 EP mk_element( int i, char c, EP next )
21    {
22       EP p = malloc( sizeof( ELEMENT ) );
23       if ( p != 0 )             // malloc() war erfolgreich?
24       {
25          p->next = next;        // Initialisierung
26          p->data.ivalue = i; p->data.cvalue = c;
27       }
28       return p;                 // Adresse des neuen Elementes
29    }
30
31 int main( int argc, char **argv )
32     {
33        EP first = 0, p;
34        first = mk_element( 3, 'c', first );
35        first = mk_element( 2, 'b', first );
36        first = mk_element( 1, 'a', first );
37        printf( "Liste:\n" );
38        for( p = first; p != 0; p = p->next )
39           prt_data( & p->data );
40     }
```

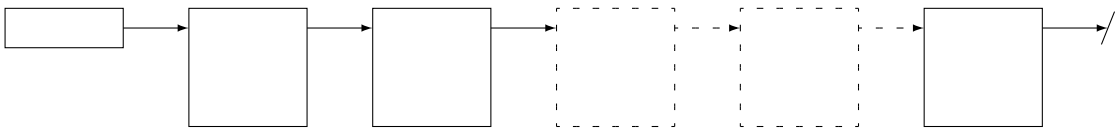
Kapitel 73

Systematik von Listen

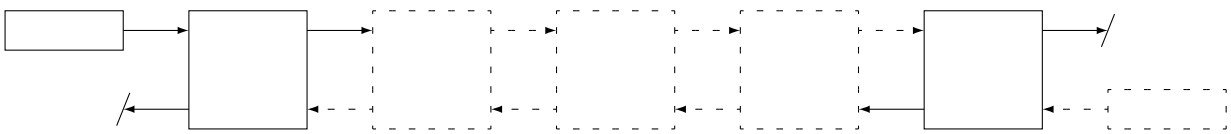
Das vorherige Kapitel hat das Konzept der Listen quasi mit einem Sprung ins kalte Wasser eingeführt; irgendwie muss man ja mal damit anfangen. In diesem Kapitel wollen wir kurz eine Systematik für diese Form der Listen einführen. Es ist nämlich sehr wichtig, dass man zwischen ihrer Struktur und ihrer Organisation unterscheidet.

73.1 Struktur

Einfach verkettete Listen: Bei den in Kapitel 72 eingeführten Listen handelt es sich um *einfach verkettete* Listen, auch lineare Listen genannt, da es von einem Element immer nur einen Zeiger auf das nächste Element gibt. Dies ist in vielen Fällen völlig ausreichend. Die kanonische Darstellungsform dieses Listentyps sieht wie folgt aus:

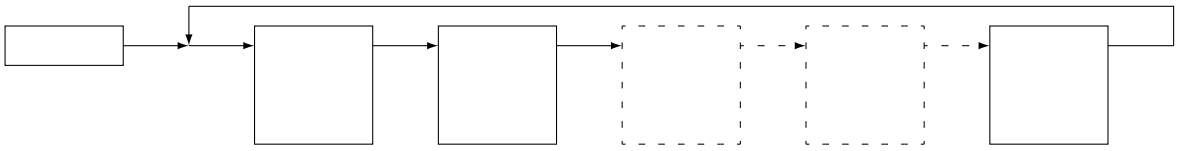


Doppelt verkettete Listen: Ein in manchen Fällen gravierender Nachteil der einfach verketteten Listen ist jedoch, dass man nicht einfach ein Element zurückgehen kann; man muss wieder vorne anfangen und so lange Element für Element voranschreiten, bis man an der gewünschten Stelle ist. Um diesen Nachteil zu beheben, gibt es auch doppelt verkettete Listen. Nein, hier zeigen nicht zwei Zeiger von einem Element zum nächsten, sondern ein Zeiger zeigt zum nächsten, ein anderer zum vorherigen. Grafisch sieht dies wie folgt aus:



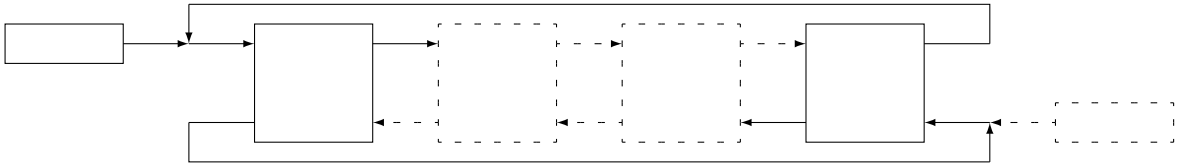
Doppelt verkettete Listen haben verschiedene Vorteile. Man kann sich einfach nach vorne und hinten bewegen und von jedem Element aus (mit Ausnahme des ersten und letzten) kann man auf sich selbst kommen (einmal vor und einmal zurück oder umgekehrt).

Einfach verkettete, zyklische Listen: In manchen Anwendungen kann es sinnvoll sein (zumindest denken dies die Programmierer), den „`next`“-Zeiger des letzten Elementes wieder auf das erste Element zeigen zu lassen. Dies sieht dann wie folgt aus:



Allerdings sollte man bei zyklischen Listen beachten, dass bei derartigen Listen die Gefahr groß ist, dass ein Algorithmus in eine Endlosschleife gerät!

Doppelt verkettete, zyklische Listen: Wenn man bei einer doppelt verketteten Liste auch den „Rückwärtszeiger“ wieder nach hinten verbindet, haben wir eine doppelt verkettete zyklische Liste, was wie folgt aussieht:



Auch hier gilt wieder, dass die Gefahr in eine Endlosschleife zu geraten, recht groß ist.

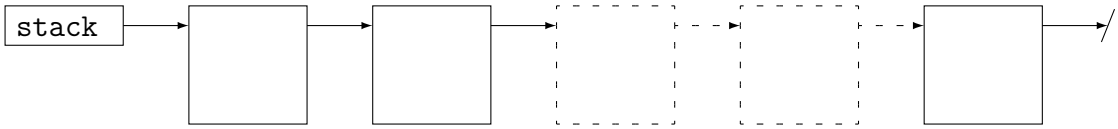
Weitere Formen: Natürlich kann man je nach Anwendungsfall seine Zeiger auch ganz anders machen. Dies kann man machen, wie man will. Obige vier Formen sind Grundformen, die sich in vielen Anwendungen und in vielen Lehrbüchern wiederfinden.

73.2 Organisation

Unabhängig von der oben besprochenen Verzeigerung kann eine Liste in der einen oder anderen Form organisiert werden. Auch diese Organisationsformen sind sehr verbreitet und können in jedem Lehrbuch wieder gefunden werden.

Stack, LIFO: Beide Begriffe, Stack und Last-In-First-Out (LIFO), können synonym verwendet werden. Wie wir bereits aus der Abarbeitung unserer Funktionen wissen, wird dasjenige Element zu erst entfernt (First Out), das als letztes eingetragen wurde (Last In). Der Prozessor macht dies natürlich in Hardware, was sehr schnell ist, aber wir können es auch in Software machen. Das Beispiel aus dem vorherigen Kapitel war ein Stack, und zwar

ein Stack, der mittels einer einfach verketteten Liste realisiert wurde. Ein Stack läßt sich wie folgt visualisieren:



Durch die gewählte Organisationsform ist die Reihenfolge der Einträge nicht vom Inhalt sondern vom Zeitpunkt des Einfügens abhängig. Insofern sind diese Listen in der Regel *unsortiert.* Das Hinzufügen und Entfernen geschieht immer am einzigen Zeiger, der hier `stack` heisst.

Warteschlange, FIFO: Auch diese beide Begriffe, Warteschlange und First-In-First-Out (FIFO), können synonym verwendet werden. Im Gegensatz zum Stack ist es so, dass dasjenige Element zuerst entfernt wird, das als erstes in die Liste eingetragen wurde. Dies entspricht dem normalen Erscheinungsbild einer Warteschlange, wie wir es täglich beim Bäcker erleben. Eine entsprechende Visualisierung sieht wie folgt aus:

Die Anordnung der beiden Zeiger `in` und `out` ist für viele ein wenig kontraintuitiv aber dennoch gut, wie hoffentlich in den folgenden Kapiteln klar wird. Auch diese Liste ist aufgrund ihrer Organisationsform in der Regel *unsortiert* bezüglich ihrer Inhalte. Das Hinzufügen geschieht hier immer am Zeiger `in`, wohingegen das Entfernen immer am Zeiger `out` vorgenommen wird.

Sortierte Listen: Diese Listen werden in der Regel immer als einfach verkettete Liste realisiert. Das Einfügen geschieht immer so, dass die einzelnen Elemente sortiert bleiben. Dies bedeutet, dass neue Elemente sowohl hinten angehängt als auch vorne oder an jeder beliebigen Stelle eingefügt werden können. Die Programmierung dieser Listenform gestaltet sich nicht ganz einfach, weshalb wir hierauf in Kapitel 75 dediziert eingehen.

Kapitel 74

Der Stack

Bereits in Kapitel 73 haben wir geklärt, dass ein Stack als einfach verkettete Liste realisiert und als LIFO (Last-In-First-Out) organisiert werden kann. In diesem Kapitel zeigen wir eine beispielhafte Implementierung nebst aller zugehörigen Funktionen.

74.1 Entwurf

Funktionalitäten: Bevor wir ans Programmieren gehen, lohnt sich hier zu überlegen, was für Operationen wir eigentlich benötigen. Zu einem Stack gehören in jedem Falle folgende Funktionalitäten:

`int isEmpty():`
: Mit dieser Funktion können wir überprüfen, ob etwas auf dem Stack liegt oder nicht.

`DP getFirst():`
: Diese Funktion liefert einen Zeiger auf den Datensatz des obersten Stack-Eintrages zurück. Sollte der Stack leer sein, wird ein Null-Zeiger zurückgegeben.

`int pushEntry( DP dp ):`
: Mittels der `push`-Operation wird ein neuer Datensatz auf den Stack gepackt, der anschließend zu oberst liegt. Je nach erfolgreicher Ausführung liefert die Funktion einen Rückgabewert; und „falsch“ signalisiert, dass `malloc()` nicht erfolgreich ausgeführt werden konnte. Die Übergabe geschieht mittels eines Zeigers, intern wird aber eine Kopie der Daten angelegt.

`int popEntry():`
: Diese Funktion entfernt den obersten Eintrag, sofern einer vorliegt.

Typdefinition und Variablenhaltung: Als nächstes müssen wir überlegen und entscheiden, welche Typdefinitionen wir brauchen, wo diese sichtbar sein sollen und wie wir es mit

der Verwaltung des Anfangszeigers machen. Hierzu haben wir als Lehrpersonal die beiden folgenden grundlegenden Gedanken:

1. Im Rahmen der Lehrveranstaltung und dieser begleitenden Unterlage wollen wir die Dinge möglichst einfach halten, damit Ihr die *wesentlichen* Dinge lernt.
2. Wir gehen davon aus, dass die wenigsten von Euch am Ende des Semesters mehr als einen richtigen Stack gleichzeitig in einem Programm verwalten werden.

Diese beiden Gründe veranlassen uns dazu, den Stack als „Modul“ `stack.c/stack.h` zu gestalten, wie wir es bereits in Kapitel 55 beschrieben haben. Dies hat den Vorteil, dass wir die Typdefinitionen für den Stack nicht nach außen sichtbar machen müssen; es reicht aus, dies in der Datei `stack.c` zu verbergen.

74.2 Kodierung des Stacks

Aus obigem Entwurf können wir den Stack direkt realisieren. Zunächt die Definition der Modul-Schnittstelle:

```
1  /*
2   * file:        stack.h
3   *
4   * description: a first, simple stack
5   *
6   * includes:    this module requires the file data.h that
7   *              should contain the definition of the data
8   *
9   */
10
11 int isEmpty();
12 DP  getFirst();
13 int pushEntry( DP dp );
14 int popEntry();
```

Als nächstes direkt die Implementierung, die aus Platzgründen recht kompakt ist:

```
1  /*
2   * file:        stack.c
3   *
4   * description: implementation of the first, simple stack
5   *
6   * includes:    this module requires the file data.h that
7   *              should contain the definition of the data
8   */
9
```

```
10 #include <stdlib.h>     // fuer malloc() and free()
11 #include "data.h"       // the definition of the content
12 #include "stack.h"    // our own specification for consistency
13
14 typedef struct _stack { // struct has name '_stack'
15                 struct _stack   // Adresse of the next
16                        *next;  // element (admin)
17                 DATA data;      // the actual (user) data
18         } STACK, *SP;           // type: data & pointer
19
20 static SP stack = 0;            // list head, empty
21
22 static SP mk_element( DP dp, SP next )  // static, used only
23        {                                  // here
24           SP p = malloc( sizeof( STACK ) );
25           if ( p != 0 )         // malloc() successful?
26           {
27              p->next = next; p->data = *dp;   // init and
28           }                                   // save data
29           return p;             // address of the new element
30        }
31
32 int isEmpty() { return stack == 0; }
33
34 DP  getFirst() { return isEmpty()? 0: & stack->data; }
35
36 int pushEntry( DP dp )
37     {
38        SP p = mk_element( dp, stack );  // make new entry
39        if ( p != 0 )                    // successful ?
40           stack = p;                 // add element to stack
41        return p != 0;
42     }
43
44 int popEntry()
45     {
46        SP p = stack;            // we need one help pointer
47        if ( ! isEmpty() )
48        {
49           stack = stack->next;  // remove from list
50           free( p );            // free memory
51        }
52        return 1;                // always successful
53     }
```

Als letztes noch ein einfaches Testprogramm:

```
 1 #include <stdio.h>
 2 #include "data.h"
 3 #include "stack.h"
 4
 5 int prt_data( DP dp )            // Funktion zum Drucken
 6     {
 7         printf( "\ti= %2d c= '%c'\n", dp->ivalue,dp->cvalue );
 8     }
 9
10 int main( int argc, char **argv )
11     {
12        DATA d1 = { 1, 'a' }, d2 = { 2, 'b' }, d3 = { 3, 'c' };
13        if (    pushEntry( & d1 ) && pushEntry( & d2 )
14             && pushEntry( & d3 ) && pushEntry( & d1 ) )
15           for( ; ! isEmpty(); popEntry() )
16              prt_data( getFirst() );
17        else printf( "Sorry, pushEntry() klappte nicht\n" );
18        return 0;
19     }
```

In den Zeilen 13 und 14 werden vier Einträge auf den Stack gepackt, die in den Zeilen 15 und 16 wieder ausgegeben und vom Stack abgeräumt werden. Die Ausgabe ist wie zu erwarten wie folgt:

```
1          i=  1 c= 'a'
2          i=  3 c= 'c'
3          i=  2 c= 'b'
4          i=  1 c= 'a'
```

74.3 Schlussbemerkungen

Wie oben bereits angedeutet, kann man die hier vorgestellte Realisierung des Stacks kritisieren: Durch die Wahl eines Datei-globalen Stack-Zeigers in Zeile 20 der Datei `stack.c` kann hier nur ein Stack pro Programm verwaltet werden. Möchte man mehr als einen Stack gleichzeitig haben, müsste man entweder diese Datei mehrfach haben (unter anderem Namen) oder den Stack-Pointer als Parameter an alle Funktionen übergeben. Zusätzlich müsste man die Definition des Stack-Datentyps in die Datei `stack.h` übernehmen, was auch unschön wäre. Unabhängig davon müßte man die Datei `stack.c` vervielfachen, wenn man weitere Stacks mit einer anderen Nutzdatenstruktur haben möchte. Schön wäre eine generische Lösung, die hier aber erst einmal den Rahmen sprengt, denn es geht uns hier primär um das Erlernen und Festigen einfacher Listen.

Kapitel 75

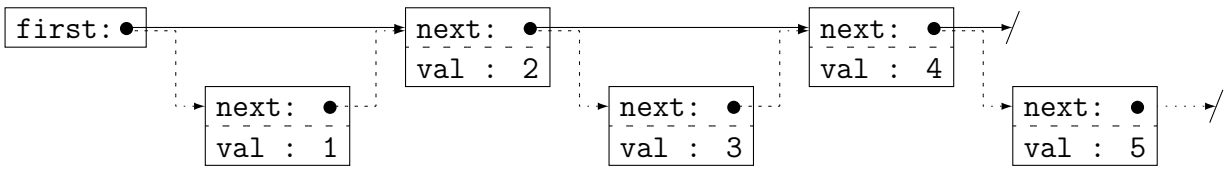
Sortierte Listen

In diesem Kapitel widmen wir uns den sortierten, einfach verketteten Listen. Diese sind *keine* Stacks mehr, denn der Ort des neuen Elementes hängt nicht vom Zeitpunkt des Einfügens sondern von seinem Wert ab. Dies macht das Auffinden des richtigen Ortes zum Einfügen notwendig. Doch in seiner Standardform ist dieses Auffinden durch mehrere Sonderfälle gekennzeichnet und daher eher unübersichtlich. Dennoch präsentieren wir diese sehr weit verbreitete Form der Verwaltung sortierter Listen. Am Ende zeigen wir noch eine Alternative. Diese entstammt ursprünglich aus [5] und führt zu einem deutlich einfacheren Algorithmus. Um die Quelltexte möglichst einfach zu halten, betrachten wir in diesem Kapitel nur den sehr einfachen Fall, dass einfache Zahlenwerte vom Typ `int` verwaltet werden.

75.1 Drei Positionen zum Einfügen

Das Schwierige an sortierten, einfach verketteten Listen ist, dass es beim Einfügen drei grundlegend verschiedene Fälle gibt. Nehmen wir an, wir hätten bereits eine Liste mit den beiden Werten `2` und `4`. Nehmen wir weiterhin an, dass wir die Elemente `1`, `3` bzw. `5` einfügen wollen. Dann müssen wir entweder vor das erste Element einfügen (und damit auch den Start-Zeiger verändern), zwischen zwei bestehende Elemente einfügen oder an das Ende der Liste anhängen. Diese drei Fälle haben wir in folgender Grafik dargestellt:

Drei Einzelfälle bei Einfügen neuer Elemente



Diese drei Fälle müssen sich natürlich im Algorithmus wiederfinden, den wir wie folgt

aufbauen können:

Fall 1: vor den Anfang einfügen:
In diesem Fall ist die Liste entweder leer, oder wir müssen das neue Element `new` vor den Anfang einfügen. Das entsprechende Code-Fragment wäre dann:

```
19 if ( first == 0 || new->val < first->val )
20 {
21    new->next = first; first = new;
22 }
```

Fall 2: mitten in die Liste einfügen
Die Schwierigkeit hier ist, dass die „normale“ Suchschleife

```
1 p = first;
2 while( p->val < new->val )
3    p = p->next;
```

ein Element zu spät stehen bleibt. Beispielsweise würde der Hilfszeiger `p` beim neuen Wert `3` auf das Element zeigen, in dem sich der Wert `4` befindet. Das ist eins zu spät und zurück geht nicht mehr. Also muss die Schleife wie folgt formuliert werden:

```
24 p = first;
25 while( p->next != 0 && p->next->val < new->val )
26    p = p->next;
```

Mit dem ersten Teil der Schleifenbedingung stellen wir sicher, dass wir nicht über das letzte Element hinaus gehen (also nicht von der Klippe springen). Mit dem zweiten Teil der Schleifenbedingung stellen wir sicher, dass wir *früh genug* stehen bleiben. Nun müssen wir noch abfragen, ob wir ein- oder anhängen müssen:

```
27 if ( p->next )
28 {
29    new->next = p->next; p->next = new;
30 }
```

Fall 3: an das Ende anhängen
Diesen Fall kann man als `else`-Teil von Fall 2 formulieren:

```
31 else { p->next = new; new->next = 0; }
```

Auf der nächsten Seite sehen wir nochmals den kompletten Algorithmus nebst Typdefinition und einem kleinen Testprogramm in kompakter Darstellung aufgrund Platzmangels.

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

```
3
4  typedef struct _list {
5                  struct _list *next;
6                  int val;
7          } LIST, *LP;            // Type Data & Pointer
8
9  LP mk_element( int val )
10     {
11        LP p = malloc( sizeof( LIST ) );
12        if ( p != 0 )
13        { p->next = 0; p->val = val; }
14        return p;
15     }
16 LP insert( LP first, LP new )
17    {
18       LP p;
19       if ( first == 0 || new->val < first->val )
20       {
21          new->next = first; first = new;
22       }
23       else {
24              p = first;
25              while( p->next != 0 && p->next->val < new->val )
26                 p = p->next;
27              if ( p->next )
28              {
29                 new->next = p->next; p->next = new;
30              }
31              else { p->next = new; new->next = 0; }
32            }
33       return first;     // in case first has changed (case 1)
34    }
35 int main( int argc, char **argv )
36     {
37        LP first = 0, p;
38        if ((p = mk_element(3))!=0 ) first = insert( first,p );
39        if ((p = mk_element(8))!=0 ) first = insert( first,p );
40        if ((p = mk_element(1))!=0 ) first = insert( first,p );
41        printf( "Die Liste:" );
42        for( p = first; p != 0; p = p->next )
43           printf( " %d", p->val );
44        printf( "\n" );
45     }
```

75.2 Alternative nach Wirth

Der oben beschriebene Algorithmus ist als Standard in den meisten Leerbüchern beschrieben. Aber die verschiedenen Sonderfälle und die Tatsache, dass sich der Startzeiger `first` ändern kann, machen den Algorithmus unhandlich und eher länglich. Wirth [5] hat eine Alternative vorgeschlagen, die insbesondere aus didaktischer Sicht recht lohnenswert ist.

Der Ansatz: Wirths Ansatz basiert im Wesentlichen auf zwei Ideen:

1. Die Liste wird am Ende um ein Dummy-Element erweitert, dass *immer* am Listenende sein wird. Dieses Element gehört technisch gesehen (aus administrativer Sicht) zur Liste, inhaltlich aber *nicht.*
2. Durch dieses zusätzliche Element können wir erreichen, dass bei der Suche, beispielsweise `while( p->val < new->val )` ein „Suchzeiger“ `p` spätestens an diesem Element stehen bleibt. In unserem Beispiel *nehmen* wir an, dass in diesem Dummy-Element die Zahl `INT_MAX` steht; da wir das weiter unten noch besprechen werden, haben wir diese Zahl nicht in die Grafik eingetragen.

Folgende Grafik zeigt, wie anfangs die Liste nur aus einem Dummy-Element besteht und wie das neue Element eingefügt wird. In der Grafik sind zusätzlich die Speicheradressen der beiden Listenelemente eingetragen.

Vor Einfügen eines neuen Elementes mit val=8

Nach Einfügen des neuen Elementes mit val=8

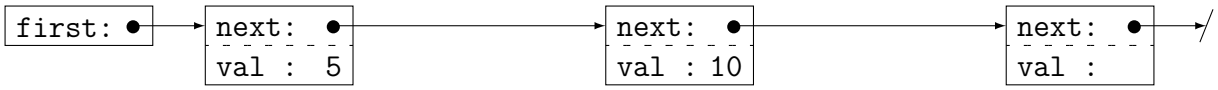
Die Grafik zeigt die folgenden vier Dinge, die spezifisch für Wirths Algorithmus sind:

1. Der „Suchzeiger“ `p` ist in der Tat ein Element zu spät stehen geblieben. „Zu spät“, da `p->val` bereits größer als `new->val` ist. Dies ist auch keine Überraschung, da im Dummy-Element *konzeptuell* die Zahl `INT_MAX` steht.
2. Das neue Listenelement wird *hinter* dasjenige Element eingefügt, auf das der Suchzeiger `p` zeigt. Durch dieses Anhängen ist die Liste jetzt erst einmal unsortiert.
3. Durch Austauschen der Inhalte der beiden Listenelemente, auf die die beiden Zeiger `p` und `new` zeigen, ist die Liste wieder sortiert. Jetzt ist alles wieder gut!
4. Durch das Tauschen der Inhalte der beiden Listenelemente ist es nicht mehr sinnvoll

vom „neuen Element“ zu sprechen. Vielmehr sollten wir von „neuen Datensatz“ (die eigentlichen Daten) und „neuen Listenelement“ (das neue Speichersegement) reden.

Illustrationsbeispiel: Die folgenden Erläuterungen illustrieren wir mit einem kleinen Beispiel, in dem eine Liste gegeben ist, die aus den Elementen `val=5` und `val=10` sowie einem Dummy-Element besteht.

Ausgangssituation: Liste mit den Werten 5 und 10 sowie Dummy-Element



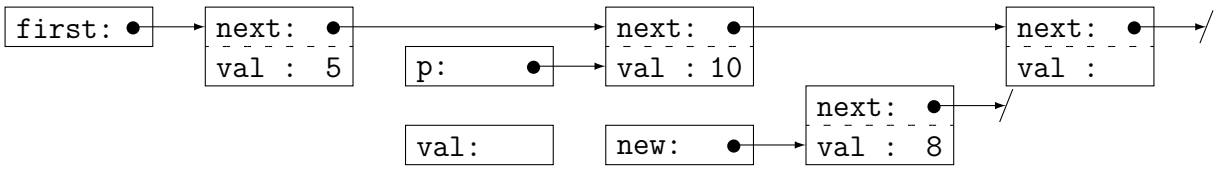
In diesem Beispiel wollen wir ein neues Element mit dem Wert `val=8` einfügen.

Implementierungsdetails: Im ersten Anlauf sind wir (konzeptuell) davon ausgegangen, dass im Dummy-Element die größt mögliche Zahl, `INT_MAX`, steht, damit die `while()`-Schleife spätestens hier terminiert. Das klappt für `int`-Zahlen recht gut. Aber derartige Konstanten sind nicht immer in einfacher Weise für alle Datentypen verfügbar. Schwierig wird es insbesondere, wenn die Nutzdaten ganze `struct`s oder andere komplexe Datentypen sind. Es kann also im Einzelfall schwierig sein, einen geeigneten Wert für das Dummy-Element zu finden. Aber ein genaues Überlegen zeigt, dass wir für das Dummy-Element gar keinen konkreten Wert benötigen, weshalb wir oben in die Grafiken auch keine konkrete Zahl eingezeichnet haben; das Beispiel mit `INT_MAX` war nur konzeptueller Natur. Das Dummy-Element ist auch durch die Eigenschaft `next == 0` charakterisiert, weshalb wir die Suchschleife auch wie folgt formulieren können:

```
20          p = first;
21          while( p->val < new->val && p->next )
22             p = p->next;
```

Wichtig zu verstehen ist, dass diese modifizierte Suchschleife in jedem Fall spätestens am Dummy-Element terminiert. Wer dies nicht sieht oder glaubt führe am besten eine Papier-und-Bleistift-Simulation durch. Für unser Beispiel ergibt sich folgendes Bild:

Zeilen 20-22: Suche nach dem nächst größeren Element; 10 in unserem Fall



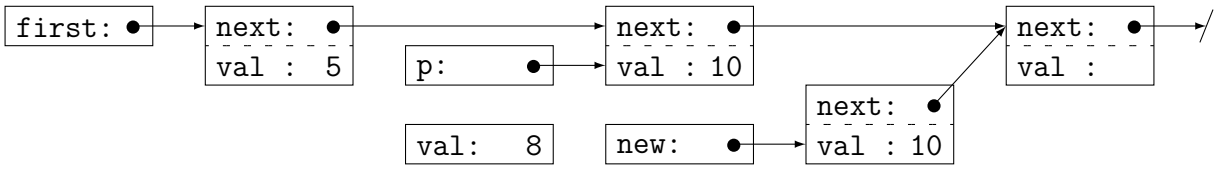
Nach dem Suchen der richtigen Stelle müssen noch die Inhalte der Listenelemente vertauscht werden, damit die Liste wieder sortiert ist. Da es sich bei den Listenelementen um normale `struct`s handelt, sollte mittlerweile jeder Lesen wissen, Zunächst sichern wir also

den Wert unsereres neuen Elementes (in unserem Fall den Wert 8) und überschreiben die Inhalte des neuen Elementes mit den Inhalten des soeben gefundenen; mit anderen Worten: wir duplizieren das soeben gefundene Element, dessen Wert größer als der Wert des neu einzufügenden Elementes ist. dies geht wie folgt:

```
23          val = new->val;    // save the data
24          *new = *p;         // copy *everything* from p to new
```

und hat in unserem Beispiel folgenden Effekt:

Zeilen 23 und 24: Sichern und Duplizieren

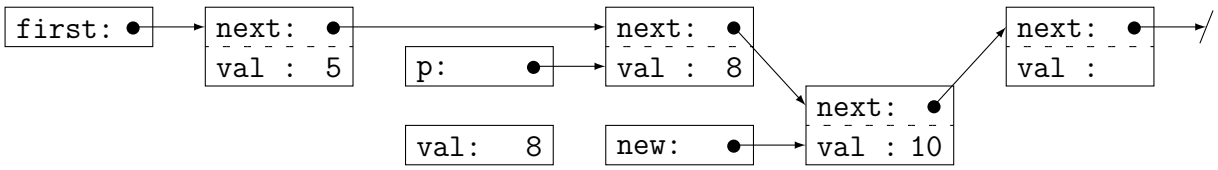


Nun müssen wir noch den gesicherten Wert zurückschreiben und das neue Element in die Liste einbinden:

```
25          p->val = val;      // bring val to old element
26          p->next = new;     // link the new element into the list
```

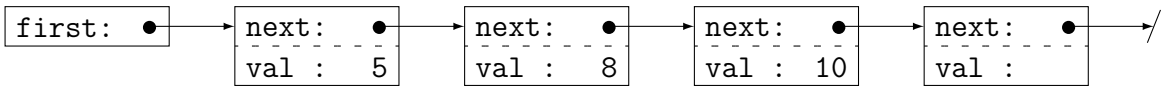
was sich wie folgt veranschaulichen läßt:

Zeilen 25 und 26: Zurückschreiben und Einbinden des neuen Elementes



Zum Schluss können wir die Listenelemente ein wenig zurechtrücken und die lokalen (Hilfs-) Variablen entfernen, was zu folgendem Bild führt:

Ergebnis der Einfügeoperation: eine Liste mit den Elementen 5, 8 und 10



Auf der nächsten Seite fassen wir nochmals alle Programmteile zu einem kompletten Beispielprogramm zusammen.

Ein komplettes Beispielprogramm: Eine mögliche, vollständige Implementierung kann wie folgt aussehen:

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 typedef struct _list {
 5                 struct _list *next;
 6                 int val;
 7         } LIST, *LP;            // Type Data & Pointer
 8
 9 LP mk_element( int val )
10     {
11        LP p = malloc( sizeof( LIST ) );
12        if ( p != 0 )
13        { p->next = 0; p->val = val; }
14        return p;
15     }
16
17 int insert( LP first, LP new )
18     {
19        int val;
20        LP  p = first;
21        while( p->val < new->val && p->next )
22           p = p->next;
23        val = new->val;   // save the data
24        *new = *p;        // copy *everything* from p to new
25        p->val = val;     // bring val to old element
26        p->next = new;    // link the new element into the list
27        return 1;
28     }
29
30 int main( int argc, char **argv )
31     {
32        LP p, first = mk_element( -1 ); // the dummy element
33        if ((p = mk_element( 3 )) != 0 ) insert( first, p );
34        if ((p = mk_element( 8 )) != 0 ) insert( first, p );
35        if ((p = mk_element( 1 )) != 0 ) insert( first, p );
36        printf( "Die Liste:" );
37        for( p = first; p->next != 0; p = p->next )
38           printf( " %d", p->val );
39        printf( "\n" );
40     }
```

Die Ausgabe lautet: `Die Liste: 1 3 8.`

Weitere Anmerkungen zu Wirths Algorithmus: Ein weiterer Vorteil von Wirths Ansatz ist, dass der Zeiger auf das erste Element nie verändert wird, was den Algorithmus

zusätzlich vereinfacht. Ferner wäre es dadurch möglich, den Aufruf zum Alloziieren des neuen Elementes in die Funktion `insert()` zu integrieren.

Eine häufige Kritik an Wirths Algorithmus ist, dass er ein Element zu viel benötigt und somit Speicherplatz „verschwendet." Allerding sind die Einsparungen durch den verkürzten Algorithmus (kleineres Text-Segment) deutlich größer.

Ein zweiter, häufig geäußerter Kritikpunkt ist, dass unnötig viel kopiert wird, wenn die Datenelemente größer werden. Aber das sind nur Scheinargumente. Erstens ändert sich durch das Kopieren die strukturelle Komplexität des Algorithmus nicht; alle Operationen haben weiterhin lineare Laufzeiten, sind also in $O(n)$ (siehe auch nächster Abschnitt). Zweitens kann man das Kopieren sehr einfach halten, wenn man die Daten nicht in die `struct`s integriert, sondern nur einen Zeiger auf die Daten verwaltet. Ein derartiges `struct` könnte wie folgt aussehen:

```
1 typedef struct _list {
2                   struct _list *next;     // next element
3                   DATA *dp;               // pointer to data
4         } LIST, *LP;
```

75.3 Rechenzeit

Bei Sortier- und Suchalgorithmen ist es immer interessant die Rechenzeit $t(n)$ in Abhängigkeit der Zahl n der Elemente zu bestimmen. Sollte das Element nicht in der Liste vorhanden sein, so wird die Liste in n Schritten durchgegangen. Dies bezeichnet man auch als $t(n) = O(n)$. In den Fällen, in denen das gesuchte Element vorhanden ist, werden im statistischen Durchschnitt $n/2$ Schritte benötigt, was ebenfalls als $t(n) = O(n)$ bezeichnet wird. Kurz gefasst würde man sagen: Die Rechenzeit ist linear in der Zahl n der Listenelemente. Ob dies auch bei Bäumen und Hash-Tabellen so ist, sehen wir gleich.

Kapitel 76

Bäume

Nein, hier geht es weder um Apfel- noch um Kirschbäume sondern um Bäume in der Informatik. Und in der Informatik wachsen die Bäume auch nicht in den Himmel sondern in die Hölle. *„Ha ha, wer soll denn diesen Unfug glauben? Jedes Kind weiß doch, dass Bäume von unten nach oben Richtung Himmel wachsen.“* Ja, aber Kinder besuchen keine Vorlesungen in Informatik und hier wachsen die Bäume tatsächlich von oben nach unten. Wie so einiges andere auch, ist in der Informatik vieles spiegelverkehrt. Die Adressen des Arbeitsspeichers wachsen ja auch von unten nach oben, also genau anders herum, als wir es gewohnt sind. Vielleicht kommen ja die wichtigsten Informatiker von der Südhalbkugel ...

Ok, Spaß beiseite. In diesem Kapitel beschäftigen wir uns mit *binären Bäumen,* in denen, wie der Name *bi* andeutet, jeder (Ast-) Knoten zwei Nachfolger hat. *„Wie jetzt, gleich zwei Nachfolger? Ich hatte schon bei den Listen, die nur einen Nachfolger oder Vorgänger haben, meine liebe Mühe irgendwie hinterher zu kommen. Und jetzt soll ich gleich bei zwei Nachfolgern den Überblick behalten? Da weiß man doch nach ein oder zwei Knoten überhaupt nicht mehr, was wo zu finden ist. Nee, nee, nee ...* Wir wissen ja nicht, was dein Gesundheitsberater empfiehlt, aber wir empfehlen: „immer locker bleiben.“ Mit unserem bisherigen Vorwissen und ein klein wenig zusätzlicher Systematik sind diese binären Bäume gar nicht so schwer zu verstehen und sogar ein bisschen cool :-)

76.1 Die Struktur eines binären Baums

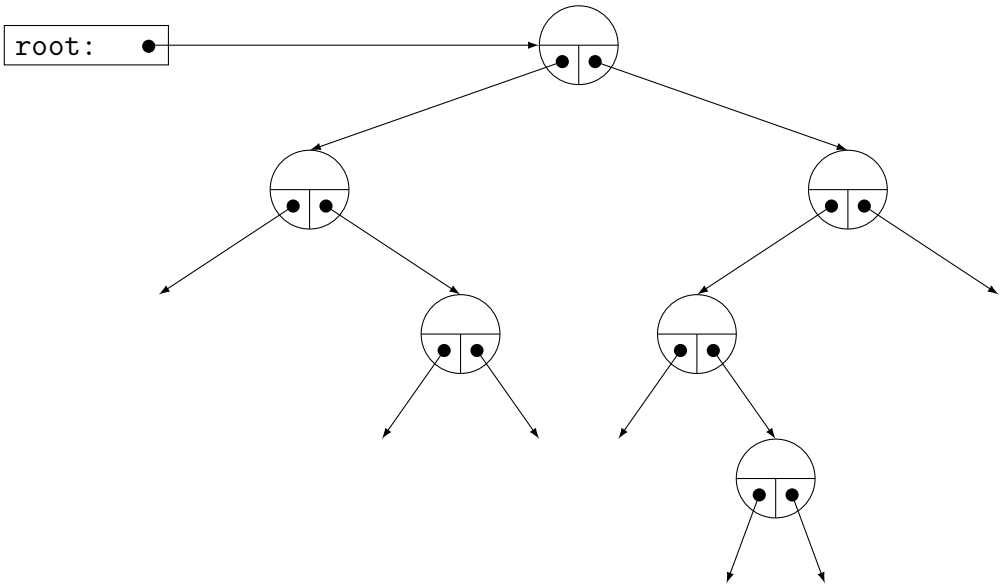
Der Einfachheit nehmen wir auch in diesem Kapitel wieder durchgängig an, dass wir nur einfache Zahlen vom Typ `int` in unseren Bäumen speichern wollen. Eine Erweiterung auf andere Datentypen sollten nach allem bisher Gelernten recht einfach sein, was in den Übungen nochmals aufgegriffen wird.

Wie bei den Listen auch müssen wir als erstes einen geeigneten Datentyp definieren. Dies geht beispielsweise wie folgt:

```
1 typedef struct _node {
2                  struct _node *left;     // linker Ast
3                  struct _node *right;    // rechter Ast
4                  int value;              // Nutzdaten
5         } NODE, *NP;                     // Typ Daten & Pointer
```

Die folgende Grafik zeigt einen derartigen binären Baum. Aus Gründen der Übersichtlichkeit und aufgrund des bereits fortgeschrittenen Wissensstandes sind in diesem Baum weder Nutzinformationen noch die Namen der Zeiger eingezeichnet.

Ein binärer Baum ohne Nutzdaten



Terminiologie: Im Zusammenhang mit Bäumen benötigen wir folgende Terminologie:

Wurzel:
Ganz oben ist die Wurzel (root), bzw. der Wurzelzeiger (root bzw. root pointer). Ist der Wurzelzeiger ein Null-Zeiger, so ist der Baum leer.

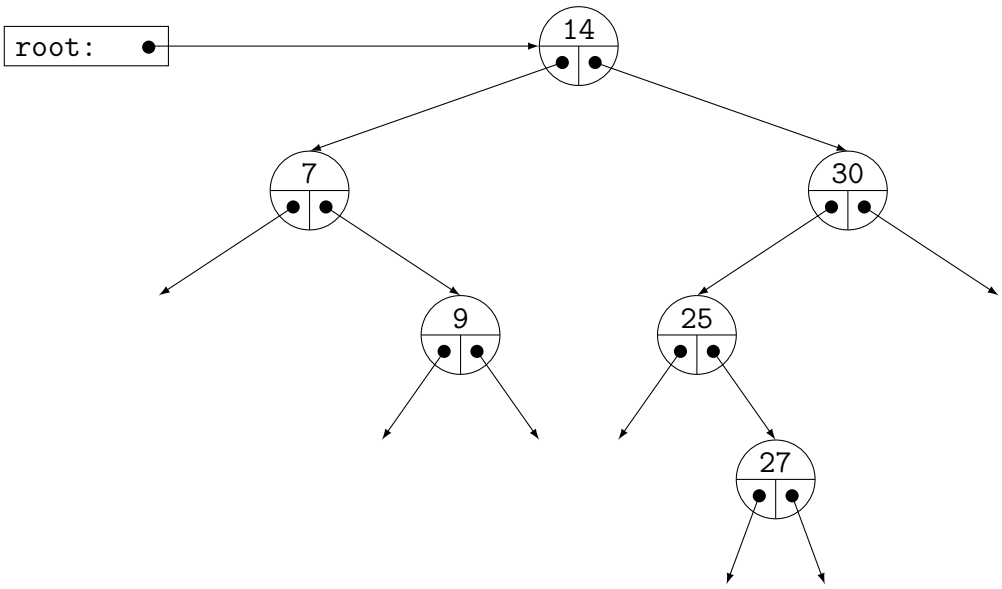
Knoten:
Außer dem Wurzelzeiger besteht der Baum aus Knoten (nodes). Jeder Knoten kann bis zu zwei Nachfolger haben, mit denen er über ein Zeiger verknüpft ist.

Blätter:
Alle Knoten, die keinen Nachfolger haben, werden auch als Blätter (leaves) bezeichnet. Entsprechend können Blätter keine weiteren Nachfolger haben.

76.2 L-K-R Sortierungen

Um nochmals die eingangs gemachte Bemerkung bezüglich des Überblickbehaltens aufzugreifen: Der Volksmund würde jetzt sagen, dass ohne eine zusätzliche Systematik binäre Bäume tatsächlich so chaotisch und unübersichtlich wären, wie übergroße, unaufgeräumte Frauenhandtaschen[1]. Eine der möglichen Sortierungen heißt L-K-R (*engl.* L-N-R) und steht für Links-Knoten-Rechts (*engl.* left-node-right). In derartigen Bäumen ist der Inhalt des linken Teilbaums immer und überall kleiner als der Inhalt des Knotens und der Inhalt des rechten Teilbaums immer und überall größer als der Inhalt des Knotens. Am besten schauen wir uns wieder ein Beispiel an, für das wir ja oben bereits die Typdefinition haben. Der folgende Baum beherbergt die Zahlen `7`, `9`, `14`, `25`, `27` und `30`.

Ein binärer Baum mit den Nutzdaten: 7, 9, 14, 25, 27, 30

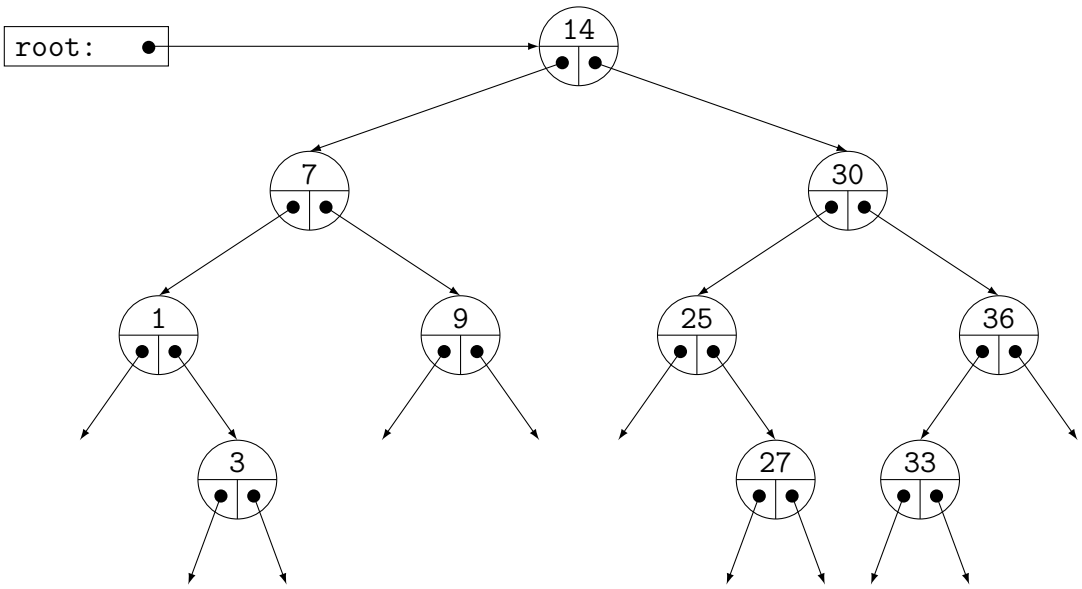


Wenn wir jetzt nochmals genau hinsehen, bermerken wir, dass obiges L-K-R-Ordnungsprinzip immer und überall eingehalten wurde. Schauen wir uns beispielsweise den Knoten mit der `14` an. Auf seiner linken Seite befinden sich nur die Zahlen `7` und `9`, die alle kleiner als `14` sind. Auf seiner rechten Seite befinden sich die Zahlen `25`, `27` und `30`, die alle größer sind.

Als nächsten schauen wir, wie weitere Zahlen eingefügt werden. Für jede neue Zahl benötigen wir natürlich einen neuen Knoten, den wir anschließend an der *richtigen* Stelle in den Baum einfügen müssen. Im folgenden Bild haben wir die Zahlen `1`, `3`, `36` und `33` der Reihe nach entsprechend eingefügt.

[1] Die Autoren distanzieren sich natürlich mit allem Nachdruck und aller Entschlossenheit von dieser volkstümlichen Weisheit, da keiner von ihnen jemals eine unaufgeräumte Frauenhandtasche zu Gesicht bekommen hat, in der Frau länger als 1,83 Sekunden nach etwas suchen musste.

Ein binärer Baum mit den Ergänzungen 1, 3, 36, 33



Bei genauem Hinsehen würde jetzt auffallen, dass das Resultat von der Reihenfolge der Ergänzungen abhängig ist. Kämen beispielsweise die beiden Zahlen `1` und `3` in umgekehrter Reihenfolge, stünde im Knoten mit der `1` eine `3` und der Knoten mit der `1` stünde links unterhalb der `3`.

„Ok, das war jetzt wirklich gar nicht so schwer. Aber was habe ich davon? Wie soll ich so einen Baum drucken? Ich kann doch nicht einfach irgendeinem Zeiger nachgehen, wie soll ich denn zurückfinden? Auch das ist gar nicht so schwer, wie wir gleich sehen werden.

76.3 Drucken eines Baums

Da in einem Baum ein Knoten in der Regel zwei Nachfolger hat, können wir tatsächlich nicht einfach einem Zeiger nachgehen, das würde nur schiefgehen. Aber wir können uns das gewählte Ordnungsprinzip zu Nutze machen: Bei einem L-K-R Baum sind ja alle Knoten im linken Teilbaum kleiner als der betrachtete Knoten und alle im rechten Teilbaum größer. Also muss man nur zuerst immer den linken Teilbaum komplett ausgeben, dann den betrachteten Knoten und dann seinen rechten Teilbaum. Dieses Prinzip gilt für jeden Knoten dieses Baumes.

Genau, das geht am besten rekursiv. Spätestens beim Null-Zeiger müssen wir aufhören, denn das Dereferenzieren eines derartigen Zeigers führt bekanntlich zum Programmabsturz. Da bereits der komplette Baum leer sein kann (der `root`-Zeiger wäre in diesem Falle ein Null-Zeiger) wäre das bereits ein geeignetes Abbruchkriterium. Der Quelltext dazu ist super simpel:

```
1 void print_tree( NP root )
2      {
3         if ( root )     // we have something
4         {
5            print_tree( root->left );
6            printf( " %d\n", root->value );
7            print_tree( root->right );
8         }
9      }
```

„Ok, das ist wirklich simpel! So langsam gefällt es mir, beinahe ein bisschen cool. Aber was ist mit dem Aufbau eines derartigen Baums? Mir scheint es doch recht schwer, den richtigen Knoten und den richtigen Zeiger auszuwählen.“ Ja, das ist leider nicht ganz so einfach aber machbar, wie wir gleich sehen werden.

76.4 Sortiertes Einfügen neuer Knoten

Für das Einfügen neuer Knoten gibt es prinzipiell zwei verschiedene Ansätze. Der erste Ansatz ist recht klassisch mit Doppelzeigern aufgebaut und der wohl gebräuchlichste. Ein alternativer Ansatz verwendet keine Doppelzeiger, verändert dafür aber den alten Knotenzeiger über den Rückgabewert der `insert()`-Funktion.

Gebräuchlicher Ansatz: Die wohl üblichste Form des Einfügens arbeitet wie folgt: Zuerst gehen wir einfach mal den Baum *rekursiv* abwärts, bis wir einen Null-Zeiger gefunden haben. Diesen können wir aber nicht mehr so einfach ändern. Entweder müssen wir vorher abfragen, was durch den Sonderfall des `root`-Zeigers unschön ist, oder wir übergeben nicht die Zeiger sondern Adressen auf die Zeiger. Der zugehörige Quelltext sieht wie folgt aus:

```
1 void insert( NP *root, NP new )
2      {
3         NP node = *root;            // just to make it simpler
4         if ( ! node )
5            *root = new;          // change root to new element
6         else if ( new->value < node->value )   // to the left?
7                  insert( & node->left, new );
8         else insert( & node->right, new ); // ok, to the right
9      }
```

Bevor wir weitermachen, eine Bemerkung: Die Hilfsvariable `node` in Zeile 3 dient nur der Vereinfachung des übrigen Quelltextes. Würden wir auf diese Variable verzichten, müssten wir überall `node` durch (*root) ersetzt, was auf Dauer nicht so prickelnd ist. Wenn wir uns jetzt den Quelltext der doch gar nicht so komplexen Funktion `insert()` nochmals genauer anschauen und uns an den `?:`-Operator erinnern, könnten wir auch wie folgt komprimieren:

```
1  void insert( NP *root, NP new )
2       {
3          NP node = *root;
4          if ( ! node )
5             *root = new;
6          else insert( (new->value < node->value)?
7                       & node->left: & node->right, new );
8       }
```

Alternativer Ansatz: Die oben angesprochene Alternative verwendet keinen Doppelzeiger sondern überschreibt einen ausgewählten Zeiger durch den Funktionswert. Konkret ist dies entweder der alte oder der neue Zeiger: `ptr = insert( ptr, new )`. Angewendet auf das Einfügen in binären Bäumen führt dies zu folgender `insert()`-Funktion, die wie folgt aufgerufen werden muss: `root = insert( root, new )`:

```
1  NP insert( NP root, NP new )
2     {
3        if ( ! root )
4           return new;
5        if ( new->value < root->value )
6             root->left  = insert( root->left , new );
7        else root->right = insert( root->right, new );
8        return root;
9     }
```

Das vollständige Programm: befindet sich auf der nächsten Seite und produziert die Ausgabe: `Der Baum: 1 7 9 14 25 27 30.`

76.5 Rechenzeit

Die Rechenzeit, die die Suche eines Elementes in einem Baum verbraucht, ist wesentlich kürzer als bei sortierten Listen. Unter der Voraussetzung, dass der Baum einigermaßen balanciert ist (d.h. jeweils in etwa gleich viele Knoten auf der rechten und linken Seite), wird beim Abstieg um eine Ebene die Zahl der weiteren möglichen Knoten halbiert. Durch das fortlaufende Halbieren benötigt ein Suchalgorithmus im Schnitt $t(n) = O(\lg n)$ Rechenzeit, wenn der Baum n Knoten enthält. Kurz gefasst würde man sagen: Die Rechenzeit ist logarithmisch in der Zahl n der Baumknoten, was deutlich schneller als der lineare Rechenaufwand innerhalb einfach verketteter Listen ist. Wie sich die Rechenzeit bei Hash-Tabellen darstellt, sehen wir gleich.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct _node {
5                  struct _node *left;     // linker Ast
6                  struct _node *right;    // rechter Ast
7                  int value;              // Nutzdaten
8          } NODE, *NP;                    // Typ Daten & Pointer
9
10 NP mk_node( int value )
11     {
12        NP p = malloc( sizeof( NODE ) );
13        if ( p )
14        { p->value = value; p->left = p-> right = 0; }
15        return p;
16     }
17 void print_tree( NP root )
18      {
19         if ( root )     // we have something
20         {
21            print_tree( root->left );
22            printf( " %d", root->value );
23            print_tree( root->right );
24         }
25      }
26 void insert( NP *root, NP new )
27      {
28         NP node = *root;
29         if ( ! node )
30            *root = new;
31         else insert( (new->value < node->value)?
32                      & node->left: & node->right, new );
33      }
34 int main( int argc, char **argv )
35     {
36        NP root = 0, p;
37        if (p = mk_node( 14 )) insert( & root, p );
38        if (p = mk_node( 30 )) insert( & root, p );
39        if (p = mk_node(  7 )) insert( & root, p );
40        if (p = mk_node(  9 )) insert( & root, p );
41        if (p = mk_node( 25 )) insert( & root, p );
42        if (p = mk_node( 27 )) insert( & root, p );
43        if (p = mk_node(  1 )) insert( & root, p );
44        printf( "Tree:" ); print_tree( root ); printf( "\n" );
45     }
```

Kapitel 77

Hash-Tabellen

In diesem Kapitel haben wir noch einmal ein etwas schwierigeres Thema. Aber am Ende werden wir wieder sehen, dass es gar nicht so schlimm war. Hash-Tabellen sind eine sehr spezielle Form der Datenhaltung. Durch Verwenden einer geeigneten Hash-Funktion braucht man in der Regel nicht mehr zu suchen sondern kann meist quasi direkt nachschauen, ob ein Element vorhanden ist oder nicht. Diese Beschleunigung erkauft man sich aber damit, dass die Daten nicht mehr sortiert sind. *„Wie jetzt? Obwohl die Daten nicht mehr sortiert sind, soll das Suchen jetzt sogar noch schneller gehen? Ich glaube Euch ja vieles, aber trotz meiner guten Erfahrungen mit Euch hat auch mein Glaube Grenzen ...“*

77.1 Problembeschreibung und Motivation

In den vorangegangenen Kapiteln dieses Skriptteils haben wir versucht, die Daten in geeigneter Form zu sortieren und diese anschließend zielgerichtet zu suchen. Schön wäre es ja nun, wenn wir gar nicht suchen müssten, sondern mehr oder weniger direkt überprüfen könnten, ob ein Element vorhanden ist oder nicht. Aber wo suchen?

Nehmen wir an, wir hätten eine Aufgabenstellung, in der wir uns so ungefähr 1024 Zahlen „merken“ (also abspeichern) müssen. Dann bräuchten wir ein Array oder eine Liste oder einen Baum geeigneter Größe; haben wir schon alles gehabt. Wären diese Zahlen auf den Definitionsbereich 0 bis 2047 beschränkt, wäre die Problemlösung einfach. Wir könnten uns ein Array `int z[ 2048 ]` mit 2048 Elementen definieren und alle seine Elemente mittels `for( i = 0; i < sizeof( z )/sizeof( int ); i++ ) z[ i ] = 0` mit „false“ initialisieren. Beim Eintragen der Zahl `i` setzen wir das entsprechende Element `z[ i ] = 1` auf den Wert „true“. Das Suchen der Zahl `s` wäre dann super simpel: `z[ s ] != 0`.

Nehmen wir nun aber an, die Zahlen könnten aus dem Definitionsbereich -32 768 bis 32 767 kommen. Dann bräuchten wir ein Array mit 65 536 Elementen, von denen so ungefähr 1024 Elemente nur nutzbringend verwendet werden; 64 512 Elemente wären sozusagen Karteileichen. Wir würden damit etwa 85 % des belegten Speicherplatzes nicht sinnvoll nutzen.

Hinzu kommt, dass die Initialisierung eines derartig großen Arrays auch Zeit kostet. Bei den oben genannten Größen ist dies vielleicht alles noch erträglich. Aber wenn der Definitionsbereich weiter wächst, bekommen wir definitiv Probleme. Also, keine gute Lösung. Aber der Ansatz ist gar nicht schlecht, nur müssen wir ihn noch etwas verfeinern.

77.2 Lösungsansatz: die Hash-Funktion

Hash-Funktionen sind ein alter Hut der Informatik. Bei ihnen kommt vorne ein Argument rein und hinten ein Wert heraus. Das Besondere daran ist, dass sie einen großen Definitionsbereich auf einen kleinen Wertebereich abbilden. *„Jetzt weiß ich es ja ganz genau!“* Eben nicht ;-) Aber wir haben wie immer ein oder zwei kleine Beispiele parat.

Die Funktion `int hash( int z ){ return z % 2048; }` wäre ein einfaches Beispiel für obige Problemstellung, denn alle Zahlen werden auf den Bereich 0 bis 2047 abgebildet. Zur Qualität einer derartigen Hash-Funktion kommen wir weiter unten. *„Ok, für Zahlen verstehe ich das Beispiel. Aber wie soll man eine Hash-Funktion implementieren, wenn es komplizierter wird? Kann man denn einen Hash-Wert für einen Namen (Zeichenkette) berechnen?“* Ja, auch das geht, sogar recht einfach. Schaue doch einfach mal selbst:

```
1 int hash( char *name )
2     {
3         int h;
4         for( h = 0; *name; name++ )
5             h = (h + *name) % 2048;
6         return h;
7     }
```

„So einfach?“ Ja, so einfach! Auch der Compiler verwendet eine derartige Hash-Funktion um beim Übersetzen unserer Programme die Variablen- und Funktionsnamen schnell zu finden. Allerdings können wir getrost davon ausgehen, dass der Compiler eine andere Formulierung des Funktionsrumpfes verwendet.

„Aber, es könnte doch sein, dass ein Programmierer in seinem Programm zwei Variablen verwendet, die zwar unterschiedliche Namen aber den gleichen Hash-Wert haben. Sorry, dass ich wieder nörgeln muss!“ Stimmt, das kann passieren und ist auch ganz normal. Die Lösung besteht in zwei kleinen Maßnahmen: Erstens müssen wir in der Lage sein, mehr als nur ein Wert (eine Zahl, einen Namen) unter dem gleichen Hash-Wert abzuspeichern und zweitens müssen wir immer überprüfen, ob die abgelegten Werte mit den gesuchten Werten übereinstimmen. Aber dazu kommen wir in den nächsten drei Abschnitten.

77.3 Die Hash-Tabelle

Daten und Datentyp: Wie eben gesagt, müssen wir dafür sorgen, dass wir mehr als einen Eintrag zu einem Hash-Wert ablegen können. Nichts einfacher als das. Wir nehmen

ein großes Array und verwalten unter jedem Index einen kleinen Stack. Diese Tabelle muss natürlich anfangs initialisiert werden. Die entsprechenden Code-Schnipsel sehen wie folgt aus:

```
 6 typedef struct _stack {
 7                 struct _stack *next;
 8                 int val;
 9         } STACK, *SP;
10
11 #define HSIZE   32
12 SP  htab[ HSIZE ];   // init: for(i=0;i<HSIZE;i++) htab[i]=0;
```

Hinzufügen in Hash-Tabellen: Auch das sollte mittlerweile für alle recht einfach sein. Wir müssen nur ein neues Element auf den Stack legen und den Tabellenzeiger, der den Anfang des Stacks markiert, eben noch initialisieren. Das geht wie folgt:

```
13 int addEntry( SP *htab, int val )
14     {
15        int ind = hash( val );
16        SP  p   = malloc( sizeof( STACK ) );
17        if ( p != 0 )
18        {
19           p->val = val; p->next = htab[ ind ];htab[ ind ] = p;
20        }
21        return p != 0;
22     }
```

Suchen in Hash-Tabellen: Auch das sollte mittlerweile trivial sein. Der Rückgabewert ist ein Zeiger auf den entsprechenden Eintrag, sofern das gesuchte Element vorhanden ist:

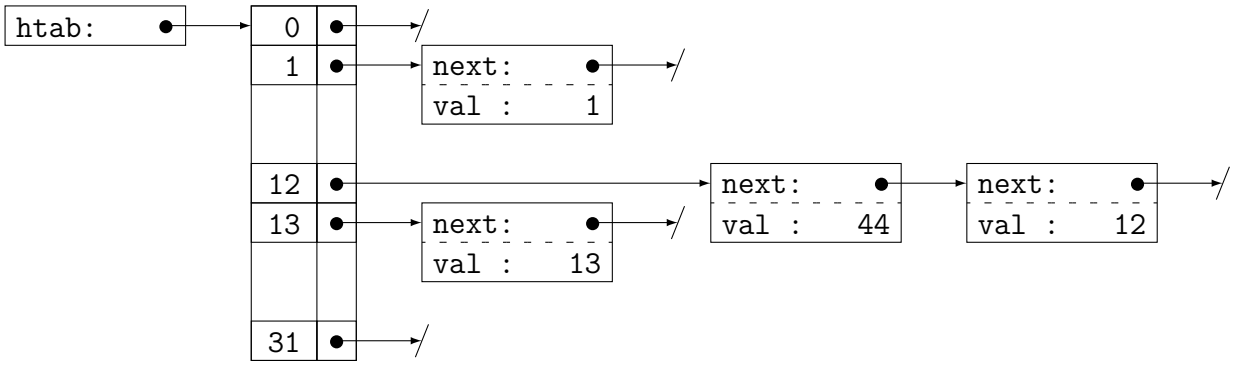
```
21 SP findEntry( SP *htab, int val )
22    {
23       SP p = htab[ hash( val ) ];
24       for( ; p; p = p->next )
25          if ( p->val == val )
26             return p;
27       return 0;
28    }
```

Das vollständige Programm: befindet sich wieder einmal in sehr komprimierter Form auf der nächsten Seite:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define HSIZE   32
```

```
5
6  typedef struct _stack {
7                  struct _stack *next;
8                  int val;
9          } STACK, *SP;
10
11 int hash( int z )
12     { return z % HSIZE; }
13 int addEntry( SP *htab, int val )
14     {
15        int ind = hash( val );
16        SP  p = malloc( sizeof( STACK ) );
17        if ( p != 0 )
18        { p->val = val; p->next = htab[ind]; htab[ind] = p; }
19        return p != 0;
20     }
21 SP findEntry( SP *htab, int val )
22    {
23       SP p = htab[ hash( val ) ];
24       for( ; p; p = p->next )
25          if ( p->val == val )
26             return p;
27       return 0;
28    }
29 int test( SP *htab, int val )
30     {
31        SP p = findEntry( htab, val );
32        printf( "%3d:%s vorhanden\n", val, p? "": " nicht" );
33     }
34 int main( int argc, char **argv )
35     {
36        int i;
37        SP  htab[ HSIZE ];
38        for( i = 0; i < HSIZE; i++ )
39           htab[ i ] = 0;
40        addEntry( htab, 12 ); addEntry( htab, 13 );
41        addEntry( htab, 44 ); addEntry( htab,  1 );
42        test( htab, 11 ); test( htab, 12 ); test( htab, 13 );
43        test( htab, 14 ); test( htab, 44 ); test( htab,  1 );
44        test( htab, 76 ); test( htab, 33 );
45     }
```

Obiges Hauptprogramm trägt die vier Einträge wie folgt in die Hash-Tabelle ein und produziert, wie zu erwarten war, folgende Ausgabe:




```
1   11: nicht vorhanden
2   12: vorhanden
3   13: vorhanden
4   14: nicht vorhanden
5   44: vorhanden
6    1: vorhanden
7   76: nicht vorhanden
8   33: nicht vorhanden
```

77.4 Rechenzeit und Wahl der Hash-Funktion

Unter der Voraussetzung einer für das Problem geeignet gewählten Hash-Funktion sind alle Elemente mehr oder weniger gleichmäßig über das Zeiger-Array verteilt. Dies bedeutet, dass auch die einzelnen Stacks leer oder nur mit einem oder vielleicht zwei Elementen besetzt sind. Das bedeutet ferner, dass nach dem Berechnen des Hash-Wertes die Suche auf einige wenige Vergleiche innerhalb eines Stacks beschränkt bleibt. Dies ist unabhängig von der Zahl der Elemente, die sich in der Hash-Tabelle befinden. Die Rechenzeit ist somit: $t(n) = O(1)$. Kurzgefasst würde man sagen: Die Rechenzeit ist konstant. Ein besseres Ergebnis für die Laufzeit kann man nicht erreichen.

Es sollte klar geworden sein, dass die Effizienz der Hash-Tabellen von einer geeigneten Wahl der Hash-Funktion abhängt. Bei einer ungeeigneten Wahl kann es passieren, dass alle Elemente in ein und denselben Stack eingetragen werden, was eine Suchzeit von $t(n) = O(n)$ nach sich zieht. Leider kann man hier keine allgemeingültigen Empfehlungen aussprechen, außer dass die Hash-Tabelle etwa doppelt so groß sein sollte, wie die Zahl der erwarteten Einträge. Ansonsten gilt: Alles hängt immer von der Anwendung ab. Man sollte sich vorher darüber Gedanken manchen, was für Elemente man erwartet, welche Werte sie haben und wie sie verteilt sein werden. Dann sollte man nach bestem Wissen und Gewissen eine Hash-Funktion formulieren, und die Effizienz testen und gegebenenfalls optimieren.

Teil VIII

Low-Level und Hardware-nahe Programmierung

Kapitel 78

Inhalte dieses Skriptteils

Eigentlich sind wir mit dem Stoff im Wesentlichen durch, obwohl wir noch nicht alle Sprachelemente besprochen haben. Das klingt ein wenig nach einem Widerspruch. Aber das, was noch fehlt, hat viel mit der Hardware-nahen Programmierung zu tun. Diese fehlenden Elemente werden in der Regel erst während Studien- oder Diplom- bzw. Bachelor- oder Masterarbeit wichtig. Manchmal braucht man diese Elemente auch erst während seiner Doktorarbeit oder vielfach nie in seinem Leben. Viele Profis jedoch setzen einige dieser Elemente ein, um ihre Programme möglichst effizient zu gestalten, zumindest gehen sie davon aus, dass dies so ist.

Über das Thema Effizienz haben wir schon kurz in Kapitel 34 gesprochen. Der wesentliche Punkt ist, dass Effizienz nicht die oberste Priorität haben sollte. Erst einmal muss das Programm richtig funktionieren. Danach kann man über Optimierungen nachdenken. Im Zuge der Optimierungen sollte man zuerst einmal Probleme des Entwurfs aufdecken und korrigieren. Die Effizienzsteigerung mittels besonders trickreicher Sprachelemente kommt ganz zum Schluss.

Aber, als Elektrotechniker muss man früher oder später auch Hardware-nah programmieren. Typische Anwendungsfälle sind das Auslesen oder Beschreiben spezieller Hardware-Register eines externen Bausteins, das Setzen oder Lesen einzelner Datenleitungen oder auch das Anpassen von Zahlendarstellungen. Für all diese Operationen stellt die Programmiersprache C geeignete Konstrukte zur Verfügung. Es geht nun wie folgt weiter:

Kapitel	Inhalt
79	Interne Repräsentation von Zahlen
80	Datentypen auf Bit-Ebene
81	Bit-Operationen
82	Ansprechen von Geräte-Registern
83	Der Datentyp `union`
84	Bit-Felder

Wir als Lehrkörper sind uns sehr darüber bewusst, dass Ihr den hier behandelten Stoff in den nächsten Semestern nicht einsetzen und damit vermutlich größtenteils wieder vergessen werdet. Daher ist dieser Skriptteil auch nicht prüfungsrelevant und wird den meisten von Euch eher als Nachschlagewerk für zukünftige Anwendungen dienen.

Aber wie immer: Schaut Euch doch einfach mal den Stoff an, probiert es und seht, dass die Dinge doch wieder einmal nicht so schwer sind.

Kapitel 79

Interne Repräsentation von Zahlen

Mit diesem Kapitel fangen wir den Abstieg in die Tiefen der Prozessorbits mit der Frage an, wie eigentlich Zahlen im Rechner dargestellt werden. Bisher haben wir uns dies nie so genau angeschaut, weil wir die konkrete Bit-Repräsentation nie benötigten. Eigentlich benötigt man sie auch nie. Aber dieses Wissen erweitert unseren Horizont und ist eine gute Vorübung für die nächsten Kapitel.

79.1 Vorbemerkungen

Bevor wir nun anfangen, hier ein paar wichtige Vorbemerkungen:

1. Um den Überblick zu behalten, verwenden wir nur acht Bits je Zahl, denn dies reicht für das Besprechen der wesentlichen Aspekte aus.
2. Wie bei der von uns Menschen gewählten Darstellung sind die niederwertigen Ziffern rechts, die höherwertigen Ziffern links.
3. Sofern nichts anderes gesagt wurde, betrachten wir die Darstellung einer Zahl immer so, wie sie im Prozessor-Register vorgenommen wird. Diese ist bei allen bekannten Prozessoren so. Im Arbeitsspeicher hingegen kann vom Prozessor eine andere Zahlendarstellung gewählt werden.

79.2 Ganze Zahlen

Positive ganze Zahlen: Von Anbeginn wurden positive ganze Zahlen so dargestellt, wie wir es von unseren eigenen Zahlen kennen, nur das die Basis nicht zehn sondern zwei ist. Um die Sache nicht unnötig zu theoretisieren, hier einfach mal ein paar Beispiele, in denen zur Veranschaulichung die Bits in zwei Vierergruppen unterteilt sind:

Zahlenwert	Bit-Repräsentation	Komposition
1	0 0 0 0 ‖ 0 0 0 1	$2^0 = 1$
35	0 0 1 0 ‖ 0 0 1 1	$2^5 + 2^1 + 2^0 = 32 + 2 + 1 = 35$
69	0 1 0 0 ‖ 0 1 0 1	$2^6 + 2^2 + 2^0 = 64 + 3 + 1 = 69$

Negative ganze Zahlen: Natürlich muss es auch die Möglichkeit geben, negative ganze Zahlen darzustellen. In der heutigen Zeit werden negative ganze Zahlen ausschließlich im 2er-Komplement dargestellt. Kurzgefasst ist bei negativen Zahlen, die im 2er-Komplement dargestellt werden, alles anders herum und zusätzlich wird auch noch eine eins addiert. Der Grund für diese anfänglich merkwürdig anmutende Kodierung ist, dass sie sich besonders gut und effizient in Hardware umsetzen lässt, worum es schließlich geht. Um nicht zu sehr zu theoretisieren, hier erst mal wieder drei Beispiele:

Zahlenwert	Bit-Repräsentation
-1	1 1 1 1 ‖ 1 1 1 1
-3	1 1 1 1 ‖ 1 1 0 1
-128	1 0 0 0 ‖ 0 0 0 0

„Wie keine Komposition? Soll man alle negativen Zahlen auswendig lernen?“ Na ja, ganz so schlimm ist es nicht. Zur Wertbestimmung einer negativen Zahl, die im 2er-Komplement gegenen ist, gibt es zwei Möglichkeiten.

Möglichkeit 1: Summe aus positiver und negativer Zahl ist null: Negative Zahlen, die im 2er-Komplement kodiert sind, sind so konstruiert, dass die Summe aus ihnen und der entsprechenden positiven Zahl null ergibt:

Zahlenwert	Bit-Repräsentation
2	0 0 0 0 ‖ 0 0 1 0
-2	1 1 1 1 ‖ 1 1 1 0
2 + (-2) = 0 $\not{1}$	0 0 0 0 ‖ 0 0 0 0

Bei dieser Betrachtung verschwindet der zusätzliche Überlauf an der vordersten Stelle, da es kein neuntes Bit gibt. Praktisch, eigentlich.

Möglichkeit 2: Konstruktion über das 1er-Komplement: „Offiziell“ ist das 2er-Komplement wie folgt definiert: Man bildet zuerst das 1er-Komplement, in dem man alle Bits negiert (aus einer eins wird eine null und umgekehrt) und anschließend eine 1 addiert. Für das Beispiel `-6` sieht das wie folgt aus:

Zahlenwert	Bit-Repräsentation								Bemerkung
6	0	0	0	0	0	1	1	0	Komposition: $2^2 + 2^1 = 4 + 2 = 6$
	1	1	1	1	1	0	0	1	1er-Komplement, alle Bits negiert
-6	1	1	1	1	1	0	1	0	2er-Komplement=1er-Komplement+1

Die Gegenprobe gemäß Möglichkeit 1 überlassen wir dem Leser :-)

Obige Beispiele zeigen folgendes Charakteristikum: Negative Zahlen haben an vorderster Stelle eine `1`, sofern sie im 1er- oder 2er-Komplement kodiert sind. Natürlich stellt sich jetzt die Frage, was mit großen positiven Zahlen ist. Beispielsweise hätte die Zahl `128` die Kodierung `1000 0000` was identisch mit der Zahl `-128` ist, wenn diese im 2er-Komplement gegeben ist. Diese Frage klären wir in Kapitel 80.

79.3 Fließkommazahlen

Kodierung: Für Fließkommazahlen hat IEEE verschiedene Formate definiert. Alle Formate sind wie folgt aufgebaut: Vorzeichenbit Mantisse Exponent. Ein gesetztes Vorzeichenbit gibt an, dass es sich um eine negative Zahl handelt. Der Exponent wird so gewählt, dass in der Mantisse eine eins vor dem Komma steht. Beispiel aus dem Dezimalsystem: Die Zahl 0.00123 würde nach diesem Verfahren als 1.23×10^{-3} dargestellt werden.

Hinzu kommt noch, dass der Wert des Exponenten durch die Addition einer Konstanten so verschoben wird, dass nur positive Werte dargestellt werden. Beispiel: bei einem 8-Bit Exponenten würden nicht die Zahlen von -128 bis 127 verwendet, sondern `Exponent+128` abgespeichert werden. Wer hier an genaueren Informationen interessiert ist, der schaue einfach mal in die Literatur, beispielsweise in das Wiki unter [10].

Rundungsfehler: Es ist noch wichtig zu verstehen, dass durch die Mantisse auch die Auflösung der Fließkommazahlen vorgegeben ist. Bei 32-Bit `double`-Zahlen ist die Mantisse in der Regel 23 Bit breit, bei 64-Bit `double`-Zahlen 52 Bit. Durch die begrenzte Auflösung können Rundungsfehler entstehen. Beispielsweise liefert bei 4-Byte `double`-Zahlen die Additon von `1.0 + 5.421011e-20` keinen Effekt mehr. Derartige Effekte sind immer schöne Ursachen für mögliche Fehler und Endlosschleifen ;-)

Kapitel 80

Datentypen auf Bit-Ebene

Nach dem wir in Kapitel 79 geklärt haben, wie verschiedene Zahlen auf Bit-Ebene dargestellt werden, sehen wir uns hier nun die verschiedenen Möglichkeiten an, unterschiedliche Datentypen zu definieren. Bisher hatten wir nur `char`, `int` und `double`, da es sich dabei um die drei für C natürlichen Datentypen handelt. Zu ergänzen ist noch, dass es sich bei `char` zwar um einen eigenständigen Datentyp handelt, dieser aber ebenfalls ein Ganzzahltyp ist.

80.1 Datentyp-Familie `int` und `char`

Größe: Hinter dem Datentyp `int` verbirgt sich nicht nur ein einziger Datentyp sondern eine ganze Famile. Diese Familie hat sogar zwei verschiedene Dimensionen, die wir uns anschauen müssen. Die erste Dimension ist die Größe: Auf *meinem* Rechner haben wir:

Typ	`short int`	`int`	`long int`	`long long int`
Größe	2 Bytes	4 Bytes	4 Bytes	8 Bytes

In obiger Tabelle sehen wir vier verschiedene Datentypen und ihre Größen. Hier betonen wir ein weiteres Mal folgenden *wichtigen* Sachverhalt: *Die konkrete Anzahl von Bytes ist nicht definiert und hängt sowohl von der CPU als auch vom Compiler ab.* Dafür ist aber definiert, dass jeder Typ mindestens so viele Bytes belegt wie sein linker Nachbar.

Vorzeichen: Die zweite Dimension betrifft das Vorhandensein oder Nichtvorhandensein eines Vorzeichens. Jeder der obigen vier Datentypen kann auch mit dem Schlüsselwort `unsigned` ergänzt werden. Das ergibt: `unsigned short int`, `unsigned int`, `unsigned long int` und `unsigned long long int`. Die `unsigned`-Spezifikation ändert nichts an der Größe des Datentyps, aber am darstellbaren Wertebereich. Auf meinem Rechner gilt:

Typ	Größe	Wertebereich
`short int`	2 Bytes	-32 768 .. 32 767
`unsigned short int`	2 Bytes	0 .. 65 535

Die Wertebereiche der anderen Datentypen kann jeder in der Standard Datei `limits.h` nachschlagen. In Kapitel 81 werden wir noch sehen, wie man diese Werte sogar selbst ausrechnen kann.

Datentyp-Familie `char`: Da der Datentyp `char` ebenfalls ein Ganzzahltyp ist, kann man ihn als Untertyp von `int` ansehen. Analog kann man den Datentyp `char` (-128 bis 127) auch als `unsigned char` (0 bis 255) verwenden. Ungünstigerweise schreibt der aktuelle C-Standard nicht vor, ob `char` tatsächlich `signed` oder `unsigned` ist; beides ist möglich und Sache des Compilers. Sollte es tatsächlich wichtig sein, was sehr selten der Fall sein wird, muss man dies bei der Deklaration explizit angeben.

Hinweis: Nun sollte auch klar sein, wie die CPU erkennt, ob es sich bei der Bit-Kombination `1000 0000` um die Zahl `128` oder `-128` handelt. Die Antwort ist zweiteilig: Die CPU erkennt und weiß es *nicht*. Dem Compiler muss es durch die Definition des Datentyps gesagt werden, sodass er beim Übersetzen die richtigen Maschineninstruktionen auswählen kann. Durch die Auswahl der Maschineninstruktionen weiß die CPU indirekt, um was für Zahlenwerte es sich handelt.

80.2 Datentyp-Familie `double`

Bei den Fließkommazahlen gibt es die drei Typen: `float`, `double` und `long double`. Eine `unsigned`-Spezifikation ist nicht möglich. Je nach dem, welchen der drei Datentypen man nimmt, hat man eine vorgegebene Länge von Mantisse und Exponent. Auf *meinem* Rechner haben wir:

Typ	`float`	`double`	`long double`
Größe	4 Bytes	8 Bytes	12 Bytes

Beim Umwandeln von größeren zu kleineren Datentypen kann man eine Verringerung der Genauigkeit sowie des Wertebereichs erleben.

80.3 Umrechnen zwischen den Datentypen

Längenanpassung: Bereits in Kapitel 37 haben wir erläutert, dass bei gemischten Ausdrücken entweder der Compiler oder der Programmierer die beteiligten Datentypen implizit bzw. explizit anpassen muss. Bei der Konvertierung zwischen `int` und `double` war dies einfach: `int`s können einfach an `double`s zugewiesen werden, da sie mathematisch eine Untermenge sind, von `double` nach `int` werden einfach alle Nachkommastellen abgeschnitten.

Bei den vielen oben genannten Ganzzahldatentypen können wir die Konvertierung mittels folgender Regeln beschreiben:

1. `sizeof( Zieldatentyp )` $\geq$ `sizeof( Quelldatentyp )`:
   Der Quellwert wird werterhaltend erweitert. Je nach Quelldatentyp wird vorne mit Nullen oder Einsen erweitert (siehe unten).
2. `sizeof( Zieldatentyp )` $=$ `sizeof( Quelldatentyp )`:
   Die Bit-Kodierung des Quellwertes wird so übernommen, wie sie ist. Es werden keine weiteren Anpassungen vorgenommen.
3. `sizeof( Zieldatentyp )` $\leq$ `sizeof( Quelldatentyp )`:
   Es werden nur die unteren Bits des Quellwertes übernommen. Dies entspricht der Berechnung des Modulos.

Werterhaltende Erweiterung: Eine merkwürdige Formulierung, die folgendes meint: Handelt es sich bei dem Quelldatentyp um ein `unsigned`, wird vorne mit Nullen aufgefüllt. Ebenso wird bei positiven `signed` Werten verfahren. Handelt es sich aber um einen negativen `signed` Wert, wird vorne mit Einsen aufgefüllt, damit gemäß 2er-Komplement der Wert erhalten bleibt. Um einen wichtigen Aspekt des eben Gesagten nochmals zu wiederholen: Bei der Erweiterung wird immer der *Quelldatentyp* betrachtet und *nicht* der Zieldatentyp.

Wertänderung: Bei allen obigen Anpassungen kann es zu einer „gefühlten“ Wertänderung kommen. Dies liegt aber nicht immer daran, dass sich die Bits ändern, sondern dass auf die selben Bits eine andere „Interpretationsbrille“ gesetzt wird, durch die die Bits eine andere Bedeutung bekommen. Das einfachste Beispiel sind negative `signed` Zahlen versus positiver `unsigned` Zahlen.

Beispiele: Zum Schluss dieses Kapitels sollen die obigen Regeln mittels einiger Beispiele illustriert werden. Dabei werden folgende Größen der Speicherplatzbelegung *angenommen:* `sizeof(char)==1`, `sizeof(short)==2` und `sizeof(int)==4`.

1. Quelle: Typ: `short int`, Wert: `0x0102` (258)
   Bit-Repräsentation: `0000 0001 0000 0010`

   Ziel:

Typ	hex	dezimal	Bit-Repräsentation
`char`	`0x02`	2	0000 0010
`unsigned char`	`0x02`	2	0000 0010
`short int`	`0x0102`	258	0000 0001 0000 0010
`unsigned short int`	`0x0102`	258	0000 0001 0000 0010
`int`	`0x0102`	258	0...0 0000 0001 0000 0010
`unsigned int`	`0x0102`	258	0...0 0000 0001 0000 0010

2. Quelle: Typ: `short int`, Wert: `-0x0102` (-258)
   Bit-Repräsentation: `1111 1110 1111 1110`

Ziel: Typ	hex	dezimal		Bit-Repräsentation
`char`	`-0x02`	`-2`		`1111 1110`
`unsigned char`	`0xFE`	`254`		`1111 1110`
`short int`	`-0x0102`	`-258`		`1111 1110 1111 1110`
`unsigned short int`	`0xFEFE`	`65 278`		`1111 1110 1111 1110`
`int`	`-0x0102`	`-258`	`1...1`	`1111 1110 1111 1110`
`unsigned int`	`0xFFF FEFE`	`4 294 967 038`	`1...1`	`1111 1110 1111 1110`

3. Quelle: Typ: `unsigned short int`, Wert: `0x0102` (258)
   Bit-Repräsentation: `0000 0001 0000 0010`
   Ergebnis wie oben in Fall 1

Ziel: Typ	hex	dezimal		Bit-Repräsentation
`char`	`0x02`	`2`		`0000 0010`
`unsigned char`	`0x02`	`2`		`0000 0010`
`short int`	`0x0102`	`258`		`0000 0001 0000 0010`
`unsigned short int`	`0x0102`	`258`		`0000 0001 0000 0010`
`int`	`0x0102`	`258`	`0...0`	`0000 0001 0000 0010`
`unsigned int`	`0x0102`	`258`	`0...0`	`0000 0001 0000 0010`

4. Quelle: Typ: `unsigned short int`, Wert: `0xFEFE` (65 278)
   Bit-Repräsentation: `1111 1110 1111 1110`

Ziel: Typ	hex	dezimal		Bit-Repräsentation
`char`	`-0x02`	`-2`		`1111 1110`
`unsigned char`	`0xFE`	`254`		`1111 1110`
`short int`	`-0x0102`	`-258`		`1111 1110 1111 1110`
`unsigned short int`	`0xFEFE`	`65 278`		`1111 1110 1111 1110`
`int`	`0xFEFE`	`65 278`	`0...0`	`1111 1110 1111 1110`
`unsigned int`	`0xFEFE`	`65 278`	`0...0`	`1111 1110 1111 1110`

Kapitel 81

Bit-Operationen

Nachdem wir in den vorherigen Kapiteln die Grundlage für das Verständnis der Zahlendarstellung auf der Bit-Ebene gelegt haben, können wir nun die interessanteren Dinge besprechen. Hierzu zählen vor allem die diversen Bit-Operationen, mittels derer einzelne Bits miteinander verknüpft werden können. Fangen wir einfach an:

Bitweises „`und`“, der `&`-Operator:

Mittels des `&`-Operators können zwei Werte bitweise verknüpft werden. Da bei einer logischen `und`-Verknüpfung beide Operanden logisch wahr sein müssen, kann dieser Operator zum gezielten Ausblenden ausgewählter Bits verwenden werden.

Beispiel 1: Die folgenden vier Ausdrücke liefern alle den selben Wert: (1): `i % 32`, (2): `i & 0x1F`, (3): `i & 0x20-1` und (4): `i & 32-1`, wobei die Bit-Operationen wesentlich schneller ablaufen.

Beispiel 2: Mittels `i & 0x1` kann man testen, ob eine `int` Variable `i` ungerade ist oder nicht. Folgendes Beispielprogramm würde `i ist ungerade` ausgeben:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4     {
5         int i = 1;
6         printf( "i ist %sgerade\n", (i & 0x1)? "un": "" );
7     }
```

Bitweises „`oder`“, der `|`-Operator:

Mittels des `|`-Operators können zwei Werte bitweise verknüpft werden. Da es bei einer logischen `oder`-Verknüpfung ausreicht, wenn einer der beiden Operanden logisch wahr ist, kann dieser Operator zum gezielten Setzen ausgewählter Bits verwenden werden.

Beispiel: Die Anweisung `i | 0xF` setzt die unteren vier Bits der `int`-Variable `i`.

Bitweises „exklusiv-oder“, der ^-Operator:
Der ^-Operator verknüpft seine beiden Operanden bitweise mittels der logischen exklusiv-oder-Verknüpfung. Diesen Operator kann man dafür verwenden, ausgewählte Bits geziehlt zu negieren (toggle-Funktion).

Das 1er-Komplement, der ~-Operator:
Das Konzept des 1er-Komplementes haben wir bereits in Kapitel 79 kennengelernt. Mittels des ~-Operators kann man es auch innerhalb eines C-Programms selbst anwenden.

Beispiel: Der folgende Ausdruck liefert den Wert -1: (~1)+1. Die Erklärung ist recht einfach. Eine 1 hat die Bit-Repräsentation 0...0 0001. Durch Anwenden des ~-Operators erhalten wir 1...1 1110. Die Addition der Konstanten 1 überführt diese Bit-Kombination in die 2er-Komplement Darstellung der Zahl -1.

Bit-Shift nach rechts, der >>-Operator:
Der >>-Operator verschiebt seinen linken Operanden um die angegebene Zahl von Bits (der rechte Operand) nach rechts. Das Verschieben wird so durchgeführt, dass bei signed-Datentypen das vorderste Bit dupliziert und bei unsigned-Datentypen eine 0 nachgeschoben wird. Dadurch wird erreicht, dass das Vorzeichen erhalten bleibt. Neben dem eigentlichen Verschieben hat dieser Operator je Bit eine int-Division durch 2 zur Folge.

Beispiel: Die Anweisung i >> 2 verschiebt die Bit-Repräsentation des Wertes der int-Variablen i um zwei Positionen nach rechts.

Bit-Shift nach links, der <<-Operator:
Das Verhalten des <<-Operators ist wie das des >>-Operators, außer dass das Verschieben nach links durchgeführt wird. Das Verschieben wird so durchgeführt, dass von rechts immer Nullen nachgeschoben werden. Damit entspricht das Verschieben nach links einer Multiplikation mit 2 je Bit-Position.

Beispiel: Die Anweisung i << 2 entspricht einer Multiplikation mit 4.

Kurzformen:
Alle Bit-Operationen können in der Kurzform angegeben werden, wie wir es ausführlich in Kapitel 43 besprochen haben. Folgende Ausdrücke sind erlaubt: &=, |=, >>= und <<=.

Weitere Beispiele: Die folgende Tabelle stellt ein paar Ausdrücke zusammen, um das oben Besprochene noch weiter zu illustrieren:

Ausdruck	Wert
((unsigned int) ~0) >> 1	INT_MAX
~0 ^ ((unsigned int) ~0) >> 1	INT_MIN
~0 & ~((unsigned int) ~0 >> 1)	INT_MIN

Kapitel 82

Ansprechen von Geräte-Registern

In vielen modernen Rechnersystemen sind externe Geräte und Controller über den regulären Daten- und Adressbus mit der CPU verbunden. Dies hat zur Folge, dass die einzelnen Register des Gerätes bzw. Controllers wie der ganz normale Arbeitsspeicher angesprochen werden. Diese Kommunikationsform nennt man auch Memory-Mapped I/O. Die Register richtig anzusprechen scheint auf den ersten Blick eher kompliziert, ist aber in Wirklichkeit ganz einfach.

Geräte-Register: Zuerst einmal schaut man am besten in die Dokumentation des Geräts. Dort stehen dann Dinge wie „Register `X` ist an der Adresse `A` und hat folgende Bits.“ Aus der Zahl der Bits bekommt man meistens heraus, was für ein C-Datentyp diesem Register entsprechen würde. Wenn man nun mehrere Register in einem Gerät hat, kann man diese zu einem `struct` zusammenfügen. Das könnte beispielhaft wie folgt aussehen:

```
1 typedef struct {
2                 char tx_reg;    // transmitt register
3                 char rx_reg;    // transmitt register
4                 char padding1;  // padding for addr. alignment
5                 char crtl_reg;  // control register
6         } TERMINAL;
7
8 #define INTERRUPT_CTRL  0x08    // interrupt control bit
```

Ansprechen des Gerätes: Die einfachste Lösung besteht darin, einen Zeiger auf dieses Geräte-Interface zu definieren:

```
1 TERMINAL *term_p = 0x123456;    // just a fancy address
```

Zugriff auf das Gerät: Mit obigem Zeiger ist diese Aufgabe ganz einfach zu erledigen. Der folgende Quelltext löscht das Interrupt Control Bit und liest anschließend das Lese-Register `rx_reg` aus:

```
1 term_p->crtl_reg = term_p->crtl_reg & ~ INTERRUPT_CTRL;
2 c = term_p->rx_reg;
```

Speicherklasse `volatile`: Zum Schluss müssen wir noch eine Kleinigkeit beachten. Da der Compiler ziemlich freie Hand beim Platzieren seiner Variablen hat, kann es vorkommen, dass alle Zugriffe auf die `struct`-Felder nicht aus dem Gerät sondern den internen CPU-Registern ablaufen. Dadurch kann es sein, dass die CPU andere Werte verarbeitet, als tatsächlich im Gerät vorhanden sind. Aus diesem Grund gibt es das Schlüsselwort `volatile`. Deklariert man nun den Zeiger `term_p` als `volatile TERMINAL *term_p`, werden alle Werte *immer* aus dem Gerät und *nicht* aus den internen CPU-Registern gelesen.

Kapitel 83

Der Datentyp `union`

Der Datentyp `union` ist so ähnlich wie der Datentyp `struct`. Der große Unterschied ist aber, dass alle Elemente eines `union`s auf der selben Speicheradresse sind. *„Wie, gibt es Platz zwischen den einzelnen Bits?“* Nein, den gibt es nicht. Es ist wirklich so, der Compiler legt alle Komponenten eines `union`s an die selbe Stelle. Dies bedeutet, dass man sinnvollerweise nur eine der Komponenten nutzen kann; die anderen Komponenten liegen brach.

Syntax: `union`s werden wie `struct`s definiert, nur dass man das Schlüsselwort `struct` durch `union` ersetzt. Hier ein Beispiel:

```
1 typedef union {
2                   int    i;
3                   double d;
4                   char   c;
5         } IDC_UNION;
```

Dieser Beispieltyp hat drei Komponenten, von denen man jeweils immer nur eine verwenden kann.

Verwendung: Hier gilt alles sinngemäß wie bei `struct`s. Das beinhaltet den Zugriff auf die einzelnen Komponenten (einschließlich Zeiger) und die Zuweisungskompatibilität.

Beispiel: Da man immer nur eine der Komponenten eines `union`s verwenden kann, muss man „irgendwie“ wissen, welche der Komponenten gerade aktuell ist. In der Regel benötigt man hierfür eine weitere Komponente, sodass ein `union` meistens Teil eines `struct`s ist. Eine beispielhafte Typdefinition inklusive einer einfachen Funktion zum Drucken eines derartigen Datensatzes befindet sich auf der nächsten Seite. Bei einem vollwertigen Programm würde man in der Regel die beiden Konstanten `0` und `1`, die der Komponentenunterscheidung dienen, durch entsprechende `#define`s ersetzen.

Der Zweck derartiger Anwendungen ist meistens, den Speicherbedarf eines Programms zu verringern. Dies geht aber nur, wenn man nur eine der Alternativen verwenden will.

```
 1 #include <stdio.h>
 2
 3 typedef union {
 4                 int    i;
 5                 double d;
 6         } ID_UNION;
 7
 8 typedef struct {
 9                 int data_type;
10                 ID_UNION id;
11         } DATA;
12
13 int print_data( DATA data )
14     {
15        switch( data.data_type )
16        {
17           case 0: printf( "i= %d\n", data.id.i ); break;
18           case 1: printf( "d= %e\n", data.id.d ); break;
19        }
20     }
```

Eine Hardware-nahe Beispielanwendung: Wie schon mehrfach gesagt, kann in der Regel immer nur eine Komponente eines `union`s verwendet werden, da alle Komponenten den selben Speicherplatz belegen. Dies kann man aber ausnutzen, um beispielsweise die interne Repräsentation eines Wertes herauszufinden:

```
 1 #include <stdio.h>
 2
 3 typedef union {
 4                 char ca[ sizeof( int )];
 5                 int  i;
 6         } UD;
 7
 8 int main( int argc, char **argv )
 9     {
10        int i;
11        UD  ud;
12        ud.i = 4711;
13        printf( "Repraesentation von i= %d:", ud.i );
14        for( i = 0; i < sizeof( int ); i++ )
15           printf( " 0x%02X", ud.ca[ i ] );
16        printf( "\n" );
17     }
```

Ausgabe des Programms: `Repraesentation von i= 4711: 0x67 0x12 0x00 0x00`

Kapitel 84

Bit-Felder

Ein weiteres Element der Programmiersprache C nennt man *Bit-Felder*. Man kann sie innerhalb von `struct`s und `union`s verwenden. Das Charakteristikum dieser Bit-Felder ist, dass man die Zahl der belegten *Bits* direkt spezifizieren kann, und sie somit sehr dicht zusammenpacken kann. Ein einfaches Beispiel sieht wie folgt aus:

```
1 typedef struct {
2                   int tx_data: 4;
3                   int rx_data: 4;
4                   int control: 3;
5           } BF_EXAMPLE;
```

In diesem Beispiel werden drei Komponenten definiert, von denen die ersten beiden je vier Bits belegen und die dritte nur drei Bits. Derartige Bit-Felder kann man beispielsweise verwenden, um ein direktes Abbild eines Hardware Registers zu erhalten. Allerdings wird gemäß aktuellem C-Standard [13] nicht garantiert, in welcher Reihenfolge die Bit-Felder in der umgebenen Struktur abgelegt werden. Insofern ist der Einsatzbereich von Bit-Feldern sehr begrenzt. Sollte tatsächlich die Notwendigkeit bestehen, auf einzelne Bits eines Datenwertes zuzugreifen, ist die Verwendung der in Kapitel 81 vorgestellten Methode eher empfehlenswert.

Teil IX

Experts Only, Absolutely no Beginners and Wannabes

Kapitel 85

Funktionszeiger

„Neulich habe habe ich eine Anwendung programmiert, bei der ich mal die eine, mal die andere Funktion brauchte. Dummerweise musste ich alles doppelt programmieren, obwohl fast alles identisch war. Na ja, cut-and-paste war eine Hilfe. Abends beim Bier habe ich lange über Arrays und Funktionen nachgedacht. Die Geschichte mit den Namen ist ja beidesmal irgendwie das gleiche Konzept. Könnte man nicht . . .“ Sorry, dass wir Dich unterbrechen. Ja, man könnte! Und da wir gerade nichts besseres vor haben, können wir darüber mal reden. Was war denn so Dein Gedanke?

„Mir fiel folgendes auf: So ein Array `int a[ 10 ]` *ist ein Speicherbereich mit zehn Variablen vom Typ* `int`*. Und der Name des Arrays, hier also* `a`*, repräsentiert den Anfang, also die Adresse des ersten Elementes dieses Arrays. Das habt ihr uns ja quasi eingebläut.“* Ja, das hat uns genauso viel Geduld gekostet wie Euch :-) Aber deswegen hattest du jetzt keine schlaflose Nacht, oder? Fiel Dir noch etwas anderes auf?

„Nein, deswegen nicht. Mir fiel auf, dass wir bereits in Kapitel 8 darüber sprachen, dass wir die Adresse einer Funktion über ihren Funktionsnamen bekommen. Und das könnte man doch mal versuchen, programmtechnisch sinnvoll zu nutzen, dachte ich. Aber wie? Ich hab's versucht, aber es ging nicht . . .“

Ja, der Gedanke ist nicht schlecht. Zur Wiederholung: ein Array muss irgendwo im Arbeitsspeicher sein. Dafür kommt entweder das Data-Segment, in der Regel aber der Stack in Frage. Und richtig, der Name des Arrays repräsentiert die Anfangsadresse des Arrays. Ebenso richtig: auch eine Funktion muss irgendwo im Arbeitsspeicher sein. Das ist für uns immer das Text-Segment. Und ja, der Name der Funktion repräsentiert die Anfangsadresse der Funktion.

„Ich höre immer Adresse. Also ist das ein Zeiger!? Aber was für ein Typ und wie geht man damit um?“ Ja, das ist genau die richtige Richtung. Es sind tatsächlich Zeiger, mit denen man sogar ordentlich herumhantieren kann. Aber der Reihe nach.

85.1 Funktionen: Anfangsadresse, Aufruf und Typ

Einfache Funktionen: Wir fangen mal mit den beiden folgenden, recht komplexen Funktionen an, die erst einmal wieder völlig sinnfrei sind:

```
1  int f1( int i )
2      {
3          return i+1;
4      }
```

```
1  int *f2( int *ip, n )
2      {
3          return ip+n; // return &(ip[n]);
4      }
```

Typ der Funktionen: Schauen wir uns als nächstes mal an, was diese Funktionen eigentlich zurück geben:

Funktion	Typ	C-Deklaration
`f1()`	Funktion die ein `int` zurückgibt	`int  f1()`
`f2()`	Funktion die ein `int *` zurückgibt	`int *f2()`

Typ der Funktionsnamen: Das war jetzt bestimmt total easy. Die spannende Frage ist jetzt für uns: Was für einen Typ haben die Konstanten `f1` und `f2`? Nochmals zur Klärung: Wir wollen nicht wissen, was für Typen `f1()` und `f2()` haben, denn das haben wir oben schon locker zusammengefasst. Nein, wir interessieren uns für die Typen von `f1` und `f2`!

„Null Plan... Großes Fragezeichen...“ Ist ganz einfach, eigentlich steht schon alles oben in der Einleitung. Wir wissen, dass `f1` und `f2` die Anfangsadressen von Funktionen sind. Also müssen es Zeiger sein. *„Zeiger worauf? Zeiger auf eine Funktion? Gibt's denn so etwas?“* Genau, Zeiger auf eine Funktion. Und zwar im ersten Fall auf eine Funktion, die ein `int` zurückgibt und im zweiten Fall auf eine Funktion, die einen Zeiger auf einen `int` liefert. Wenn man jetzt noch weiß, dass die Klammern `()` stärker binden als der Stern, kommen wir zu folgender Deklaration:

Funktion	C-Deklaration
`f1`	`int  (* f1)()`
`f2`	`int *(* f2)()`

„Was soll das denn für ein Zahlensalat sein?“ Du weißt doch: don't panic! Man muss einfach von innen nach aussen lesen. Wir müssen immer beim Namen anfangen und uns dann nach rechts bzw. links nach außen hangeln:

Deklaration: `int (* f1)()`:

C-Deklaration	Bedeutung	wo weiter?
`f1`	„`f1` ist “	jetzt nach links, denn wir sind in einer `()`
`(* f1)`	„`f1` ist ein Zeiger auf“	jetzt nach rechts, denn da stehen noch `()`
`(* f1)()`	„`f1` ist ein Zeiger auf eine Funktion“	jetzt nach links, denn rechts ist nichts mehr
`int (* f1)()`	„`f1` ist ein Zeiger auf eine Funktion, die ein `int` zurückgibt“	fertig

Deklaration: `int *(* f2)()`:

C-Deklaration	Bedeutung	wo weiter?
`f2`	„`f2` ist “	jetzt nach links, denn wir sind in einer `()`
`(* f2)`	„`f2` ist ein Zeiger auf“	jetzt nach rechts, denn da stehen noch `()`
`(* f2)()`	„`f2` ist ein Zeiger auf eine Funktion“	jetzt nach links, denn rechts ist nichts mehr
`*(* f2)()`	„`f2` ist ein Zeiger auf eine Funktion, die einen Zeiger“	links steht immer noch ’was
`int *(* f2)()`	„`f2` ist ein Zeiger auf eine Funktion, die einen Zeiger auf einen `int` zurückgibt“	fertig

„Ist ja wie immer bei Euch; erst ’mal wird es komplizierter und nicht einfacher. Aber inzwischen gebe ich die Hoffnung nicht mehr auf . . .“ Genau, im folgenden Abschnitt gehen wir einen Schritt weiter. Aber vorher sollten wir noch ein paar Details ergänzen. Obige Deklarationen würden funktionieren. Denn in der Programmiersprache C gilt: Wird eine Funktion ohne Parameter deklariert bzw. definiert, kann man ihr beim Aufruf beliebige Parameter übergeben. Entsprechend sind obige Typen unvollständig. Um dem Compiler zu helfen, sollten sie wie folgt lauten:

```
1 Type: int  (* f1)( int i );
2 Type: int *(* f2)( int *i, int n );
```

Mit diesen vollständigen Deklarationen helfen wir dem Compiler falsche Funktionsaufrufe zu entdecken.

85.2 Funktionszeiger

Wenn, wie wir im vorherigen Abschnitt gelernt haben, Funktionsnamen Zeiger auf Funktionen auf irgendwas sind, dann könnte man ja auch tatsächlich derartige Zeiger definieren:

```
11 int  (*fp1)( int n );          // Ein Zeiger auf eine Funktion,
12                                 // die int liefert
13 int *(*fp2)( int a, int n );   // Ein Zeiger auf eine Funktion,
14                                 // die int * liefert
```

„Das soll's schon sein?“ Ja, das sind zwei einfache Definitionen von Zeigern auf Funktionen. Der Unterschied zu den Funktionsdeklarationen, wie wir sie oben bei `int f1()` und `int *f2()` gehabt haben, besteht im zusätzlichen Stern und den zusätzlichen Klammern. Nochmals zur Wiederholung:

1. Durch den Stern ist fp nicht eine Funktion sondern ein Zeiger auf ...
2. Und durch die rechten Klammern `()` kein Zeiger auf ein `int` sondern ein Zeiger auf eine Funktion.

Diesen Funktionszeigern kann man nun einen Wert zuweisen. Der Wert muss natürlich vom richtigen Typ sein. Also muss es sich auch um einen Zeiger auf eine Funktion ... You've got it:

```
14 fp1 = f1;        // ohne runde Klammern!
15 fp2 = f2;        // ohne runde Klammern!
```

Nun zeigen die beiden Zeiger `fp1` und `fp2` auf den Anfang der Funktion `f1()` bzw. `f2()`. Wichtig bei der Zuweisung ist, dass dort keine runden Klammern stehen. Stünden dort welche, würden die Funktionen aufgerufen werden, ohne Klammern werden nur ihre Anfangsadressen genommen und kopiert.

„Und wie rufe ich jetzt die Funktionen auf? Etwa einfach die Zeiger dereferenzieren?“ Beinahe korrekt. Aber, wie eben schon gesagt, beim Funktionsaufruf müssen die runden Klammern erscheinen. Wir könnten also einfach schreiben:

```
16 i  = (* fp1)( 1 );
17 ip = (* fp2)( a, 1 );
```

„Ist ja schon so ein bisschen cool. Kann man das auch alleine?“ Man kann, muss aber üben! Auf der nächsten Seite kommt nochmals alles in einem sinnfreien Programm zusammen. In diesem Programm macht Zeile 20 genau das selbe wie die Zeilen 16 bis 18; Zeile 20 ist nur eine wesentlich kompaktere Schreibweise ohne Hilfsvariablen. Das Programm produziert zwei Mal die Ausgabe: `i= 3 *ip= 2`.

„Danke, aber das war ja wieder mal einiges an neuem Stoff, den ich erst mal verdauen muss.“

Das vollständige Programm:

```
#include <stdio.h>

int f1( int i )
    { return i + 1; }                  // return the value i+1

int *f2( int *ip, int n )
    { return ip + n; }                 // return pointer to a[n]

int main( int argc, char **argv )
    {
       int  (* fp1)( int );            // function pointer to f1()
       int *(* fp2)( int *ip, int n ); // fnc. pointer to f2()
       int i, *ip, a[] = { 1, 2, 3 };        // some variables
       fp1 = f1;                       // set pointer to fnc. f1()
       fp2 = f2;                       // set pointer to fnc. f2()
       i  = (* fp1)( 2 );              // equivalent to i  = f1(2)
       ip = (* fp2)( a, 1 );           // equivalent to ip=f2(a,1)
       printf( "i= %d *ip= %d\n", i, *ip );   // output values
                                       // the same, but very short
       printf( "i= %d *ip= %d\n", (*fp1)(2), *((*fp2)(a,1)) );
    }
```

85.3 Beispiel: Drucken einer Funktionstabelle

„Ok, das war jetzt ja schon ganz schön. Aber so richtig realistisch war das Beispiel noch nicht.“ Stimmt, wir wollten erst mal nur die Idee und die Herangehensweise vermitteln. *„Klar, das macht ihr ja immer so. Ich wollte ja eine Art Druckprogramm schreiben, das mal die eine, mal die andere Funktion tabellarisch ausgibt. Da das Bestimmen des Funktionswertes tief unten in der Druckfunktion passiert, müßte ich beim traditionellen Ansatz alles fest verdrahten und doppelt programmieren. Also bräuchte ich doch nur einen Zeiger auf die aktuelle Funktion übergeben, oder?* Ja, Dir ist es klar geworden. Aber fangen wir für die anderen nochmals weiter vorne an, nämlich beim letzten Programm in Kapitel 32. Dort haben wir gezeigt, wie wir `double`-Werte ohne große Rundungsproblematik errechnen können. Für diese Werte müssen wir jetzt einen Funktionswert bestimmen. Diese Funktion wird nicht „fest verdrahtet“ sondern als Zeiger spezifiziert. Das ganze ist dann relativ einfach, wie wir in diesem Abschnitt sehen werden.

Ausgabe: Die Ausgabe dieses Programms sieht wie folgt aus (die Ausgabe der beiden Teile erfolgt nebeneinander):

```
1          x | sin(x)
2  ----------+------------
3  x=   0.00 | f(x)=  0.00
4  x=   0.20 | f(x)=  0.20
5  x=   0.40 | f(x)=  0.39
6  x=   0.60 | f(x)=  0.56
7  x=   0.80 | f(x)=  0.72
8  x=   1.00 | f(x)=  0.84
```

```
10         x | cos(x)
11 ----------+------------
12 x=   0.00 | f(x)=  1.00
13 x=   0.20 | f(x)=  0.98
14 x=   0.40 | f(x)=  0.92
15 x=   0.60 | f(x)=  0.83
16 x=   0.80 | f(x)=  0.70
17 x=   1.00 | f(x)=  0.54
```

Das zugehörige Programm sieht wie folgt aus. In den Zeilen 20 und 22 wird die Funktion `printFncTab()` zum Drucken der Tabelle einmal mit einem Zeiger auf die `sin()`-Funktion und einmal mit einem Zeiger auf die `cos()`-Funktion aufgerufen.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int printFncTab( double (*fnc)(double x), char *name,
5                   double x_anfang, double x_ende, int steps )
6      {
7          int i;
8          double x, dx = (x_ende - x_anfang)/steps;
9          printf( "         x | %s\n", name );
10         printf( "----------+------------\n" );
11         for( i = 0; i <= steps; i++ )
12         {
13            x = x_anfang + i * dx;
14            printf( "x= %5.2f | f(x)= %5.2f\n", x, (*fnc)(x) );
15         }
16     }
17
18 int main( int argc, char **argv )
19     {
20        printFncTab( sin, "sin(x)", 0.0, 1.0, 5 );
21        printf( "\n" );
22        printFncTab( cos, "cos(x)", 0.0, 1.0, 5 );
23     }
```

85.4 Beispiel: modulweite Funktionszeiger

Nehmen wir an, wir würden ein größeres Modul für die Simulation künstlicher Neuronaler Netze realisieren. In so einer Simulation benötigt man beispielsweise in der Berechnungsvorschrift eine Funktion, die eine Zustandsgröße berechnet. Hierzu würden wir normalerweise eine eigenständige Funktion programmieren. Für ein paar Forschungsarbeiten könnte es

nun sinnvoll sein, diese Funktion hier und da gegen eine andere Funktion auszutauschen. Weiterhin könnte es sinnvoll sein, die Funktion wieder zurückzutauschen. Eine derartige Aufgabe könnte man wie folgt lösen.

Im Folgenden präsentieren wir eine stark vereinfachte Lösung. Das Modul besteht nur aus einem Funktionszeiger, einer standardmäßigen internen Funktion `default_fnc()`, die durch Aufruf der Funktion `set_fnc()` ausgetauscht werden kann. Diese Funktion gibt den aktuellen Wert des Funktionszeigers zurück, damit dieser wieder zurückgesetzt werden kann. Die Funktion `calculate()` nimmt die eigentliche Berechnung vor, die natürlich recht komplex wäre. Innerhalb dieser Funktion wird unser Funktionszeiger verwendet. Im konkreten Beispiel ist diese Funktion natürlich äußerst einfach und komplett sinnfrei. Hier nun die einzelnen Dateien:

module.h

```
 1 // to make it easier, we define a type DFNC_PTR this
 2 // pointer type (DFNC_PTR) is a pointer to a function
 3 // returning a double and having an argument of type double
 4 typedef double (* DFNC_PTR)( double x );
 5
 6 // we need a function to set a new function inside the module
 7 DFNC_PTR set_fnc( DFNC_PTR fnc_ptr );
 8
 9 // the function that calculates the parameter
10 double calculate( double x );
```

module.c

```
 1 #include "module.h"
 2
 3 static double default_fnc(double x){ return 2.0 * x; }
 4
 5 // defining and initializing the function pointer
 6 static DFNC_PTR cur_fnc = default_fnc;
 7
 8 // setting the function pointer
 9 DFNC_PTR set_fnc( DFNC_PTR fnc_ptr )
10          {
11             DFNC_PTR old = cur_fnc;
12             cur_fnc = fnc_ptr;
13             return old;
14          }
15
16 // calc. the parameter value; totally simplified
17 double calculate( double x )
18        { return 2 * (* cur_fnc)( x ) - 1; }
```

main.c

```
 1 #include <stdio.h>
 2 #include "module.h"
 3
 4 // just a test and debug function that prints its argument
 5 // and function value
 6 static void calcPrint( double x )
 7        {
 8           printf( "x= %4.2f calculate= %4.2f\n",
 9                   x, calculate( x ) );
10        }
11
12 // our own alternative function
13 static double alt_fnc( double x )
14        {
15           return x * x;
16        }
17
18 int main( int argc, char **argv )
19     {
20        DFNC_PTR old_fnc;          // one function pointer
21        calcPrint( 3.0 );                    // test output
22                                 // change module function
23        old_fnc = set_fnc(alt_fnc);
24        calcPrint( 3.0 );                    // test output
25        set_fnc( old_fnc );             // restore function
26        calcPrint( 3.0 );                    // test output
27     }
```

Die Ausgabe des Testprogramms ist, wie zu erwarten:

```
1 x= 3.00 calculate= 11.00
2 x= 3.00 calculate= 17.00
3 x= 3.00 calculate= 11.00
```

85.5 Array mit Funktionszeigern

In diesem Abschnitt präsentieren wir ein kleines Beispiel eines Arrays `fnc_array`, in dem sich drei Funktionszeiger befinden. Dieses Array wird mit den drei Funktionen `f1()`, `f2()` und `f3()` initialisiert, die wie immer nicht besonders sinnvoll sind. Im Hauptprogramm werden diese drei Funktionen der Reihe nach aufgerufen. In der `for`-Schleife (Zeile `15`) wird die Größe des Arrays „berechnet.“ Als Test werden diese drei Funktionen mit einem Argument aufgerufen.

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

```
 2
 3 static int f1( int i )
 4        { return i + 1; }
 5
 6 static int f2( int i )
 7        { return i + 2; }
 8
 9 static int f3( int i )
10        { return i + 3; }
11
12 int main( int argc, char **argv )
13     {
14        int i, (* fnc_array[])() = { f1, f2, f3 };
15        int tsize = sizeof( fnc_array )/sizeof(int (*)());
16        for( i = 0; i < tsize; i++ )
17           printf( "f%d( %d )= %d\n", i + 1, i,
18                   (* fnc_array[i])( i ) );
19     }
```

Die Ausgabe ist einfach wieder wie zu erwarten:

```
1 f1( 0 )= 1
2 f2( 1 )= 3
3 f3( 2 )= 5
```

85.6 Zusammenfassung

Fassen wir noch einmal zusammen: Der *Name* einer Funktion repräsentiert ihre Anfangsadresse und ist somit ein *Zeigertyp*. Entsprechend sind Zeiger auf Funktionen ein Typ in C. Von diesem Typ kann man Variablen definieren und ihnen die Anfangsadresse einer existierenden Funktion zuweisen. Anschließend kann man diese Funktionen auch über soeben definierte Zeigervariablen aufrufen. Bei der Definition eines Funktionszeigers ist auf die richtige Klammersetzung zu achten; sonst hat man eine Funktion, die einen Zeiger auf ein anderes Objekt zurückgibt. Ein korrektes Beispiel ist: mittels `int (* fp)()` ist `fp` ein Zeiger auf eine Funktion, die ein `int` zurückgibt. Und durch `j = (* fp)()` wird der Variablen `j` der Rückgabewert derjenigen Funktion zugewiesen, auf die `fp` zeigt.

Kapitel 86

Iteratoren

Im vorherigen Kapitel haben wir ein wenig mit Funktionszeigern hantiert. Diese sind besonders gut geeignet, um „abstrakte“ Datentypen zu kapseln, d.h. verschiedene Implementierungsdetails hinter geeigneten Funktionsaufrufen zu verstecken. Als Resultat der Kapselung sind vielfach auch Informationen versteckt, die man sich gerne (auch zur Programmlaufzeit) anschauen möchte. Hierfür kann man das Konzept der Iteratoren verwenden. Doch zunächst entwickeln wir eine kleine Anwendung, um erst anschließend Iteratoren am Beispiel einzuführen.

86.1 Beispiel: ein kleiner Taschenrechner

Aufgabe: In diesem Abschnitt entwickeln wir einen kleinen Taschenrechner, der in der Lage ist, zwei `int` Zahlen mit einer der Grundrechenarten zu verknüpfen. Ein wesentlicher Aspekt dabei ist, dass wir die Operatoren nicht direkt anwenden (das können wir nun wirklich schon), sondern dazu Funktionszeiger verwenden. Auch diese beherrschen wir bereits seit dem vorherigen Kapitel. Ein weiterer Aspekt ist, dass wir die Funktionszeiger nicht direkt über einen Index und eine Tabelle auswählen sondern indirekt über den Namen des Operators. Beispielsweise soll die Zeichenkette `"+"` die Addition auswählen. Für den ersten Entwurf wollen wir die vier Grundrechenarten `"+"`, `"-"`, `"*"` und `"/"` realisieren.

Modulschnittstelle: Es sollte ziemlich klar sein, dass wir unseren Taschenrechner wieder in einem Modul kapseln, um möglichst viele Details an einem Ort zu haben und vor „unerlaubten“ Zugriffen zu schützen. Da unser kleiner Taschenrechner nur ganzzahlige Argumente verarbeiten soll, haben alle nach außen sichtbaren Objekte den Präfix `ic_`, was für *int calculator* stehen soll. Die Schnittstelle (die Datei `icalc.h`) befindet sich oben auf der nächsten Seite.

Kernstück des Taschenrechners ist eine Tabelle, in der jeder Eintrag aus einem Funktionsnamen, beispielsweise `"+"`, und einem entsprechenden Funktionszeiger, beispielsweise `iadd`, besteht. Kapselung bedeutet an dieser Stelle, dass die Schnittstelle nach außen möglichst

```
 1 /*
 2  *      module : icalc.h
 3  *      exports: IC_NOINT, ic_errmsg(), ic_find(),
 4  *               ic_exec(), ic_iexec()
 5  */
 6
 7 #define IC_NOINT        ((int) ~(((unsigned int) ~0) >> 1))
 8
 9 char *ic_errmsg();
10 int  (* ic_find( char *name ))( int, int );
11 char *ic_exec( char *name, int a, int b, int *res );
12 int  ic_iexec( char *name, int a, int b );
```

so ausgelegt sein sollte, dass die Tatsache, dass zur Organisation ein Array verwendet wird, nicht sichtbar sein sollte. Oder anders ausgedrückt: sollten wir statt eines Arrays beispielsweise eine Liste verwenden, sollte dies *nicht* über die Schnittstelle nach außen dringen.

Von außen können die folgende Konstante und die folgenden vier Funktionen aufgerufen werden:

1. Die Konstante `IC_NOINT` ist ein spezieller Resultatswert, der anzeigt, dass bei einer Rechenoperation ein Fehler aufgetreten ist. Diese Zahl steht möglicherweise nicht mehr als Ergebnis zur Verfügung. Dass es sich bei der Konstanten „zufällig“ um `INT_MIN` handelt, sollte nicht weiter wichtig sein.
2. Die Funktion `ic_find(char *)` dient dem Auffinden einer Funktion über ihren Namen. Beispiel: Der Aufruf `ic_find( "*" )` soll einen Zeiger auf eine Funktion zurückgeben, die zwei Zahlen miteinander multipliziert.
3. Die Funktion `ic_exec()` sucht die über ihren Namen angegebene Funktion, führt diese ggf. aus und schreibt das Resultat in das vierte Argument. Je nach dem, ob ein gültiger Name übergeben wurde oder nicht, ist der Rückgabewert der Funktion eine Fehlermeldung oder ein Null-Zeiger.
4. Die Funktion `ic_iexec()` arbeitet sehr ähnlich wie die funktion `ic_exec()`, doch sie signalisiert einen Fehler über den speziellen Ergebniswert `IC_NOINT`.
5. Mittels der Funktion `ic_errmsg()` kann jederzeit die letzte Fehlermeldung ermittelt werden.

Realisierung des Moduls: Die Realisierung dieses Moduls ist auf der nächsten Seite zu finden.

Das vollständige Modul:

```
1  /*--------- module: icalc ---------*/
2
3  #include <string.h>
4  #include "icalc.h"
5
6  static int iadd( int a, int b ) { return a + b; }
7  static int isub( int a, int b ) { return a - b; }
8  static int imul( int a, int b ) { return a * b; }
9  static int idiv( int a, int b ) { return a / b; }
10
11 typedef struct {
12                 char *ic_name;
13                 int (* ic_fnc)( int, int );
14         } ICALC, *ICP;
15 static  ICALC ftab[] = { { "+", iadd }, { "-", isub },
16                          { "*", imul }, { "/", idiv } };
17 #define TAB_SIZE        (sizeof( ftab )/sizeof( ICALC ))
18
19 static char *last_errmsg = 0;     // saving last error, if any
20
21 char *ic_errmsg() { return last_errmsg; }
22
23 int (* ic_find( char *name ))( int, int )
24     {
25         ICP p;
26         for( p = ftab; p < ftab + TAB_SIZE; p++ )
27            if ( ! strcmp( name, p->ic_name ) )
28               return p->ic_fnc;
29         return 0;
30     }
31
32 char *ic_exec( char *name, int a, int b, int *res )
33      {
34         int (* p)(int,int) = ic_find(name); // reuse ic_find()
35         if ( p && res )
36            *res = (* p)( a, b );
37         return last_errmsg = p? 0: "no such function";
38      }
39
40 int ic_iexec( char *name, int a, int b )
41     {
42        int res;                              // resuse ic_exec()
43        return ic_exec( name, a, b, & res )? IC_NOINT: res;
44     }
```

Testprogramm: Das folgende beispielhafte Testprogramm soll zeigen, wie die einzelnen Funktionen nebst Fehlerabfrage aufgerufen werden sollten.

```
 1 #include <stdio.h>
 2 #include "icalc.h"
 3
 4 int main( int argc, char **argv )
 5     {
 6         int (* p)( int, int ), res;
 7         char *cp;
 8         if (p = ic_find( "+" ))
 9            printf( "%d\n", (*p)( 2, 3 ) );
10         else printf( "sorry, function '+' not found\n" );
11         if ( cp = ic_exec( "*", 2, 3, & res ))
12            printf( "sorry, function '%s' not found\n", cp );
13         else printf( "%d\n", res );
14         printf( "%d\n", ic_iexec( "+", 7, 7 ) );
15         if ((res = ic_iexec( "=", 7, 7 )) == IC_NOINT )
16            printf( "error on =: %s\n", ic_errmsg() );
17         else printf( "%d\n", res );
18         if ( ic_exec( "xx", 2, 3, & res ))
19            printf( "error on xx: %s\n", ic_errmsg() );
20         else printf( "%d\n", res );
21     }
```

Ausgabe: Obiges Testprogramm produziert folgende Ausgabe:

```
1 5
2 6
3 14
4 error on =: no such function
5 error on xx: no such function
```

Leser, die hier irgendwie und/oder irgendwo zweifeln oder unsicher sind, sollten in jedem Fall eine Handsimulation durchführen.

86.2 Iteratoren

Motivation: So ein gekapseltes Modul verbirgt sehr viele Details. Nun könnte es für viele Anwendungen nützlich und sinnvoll sein, einen Mechanismus zu haben, der einem verschiedene interne Details sichtbar macht. Ein Beispiel wäre eine Liste aller verfügbaren Funktionen. Im Sinne der Änderungsfreundlichkeit wäre es sinnvoll, diese Liste aus der Funktionstabelle in geeigneter Weise zu extrahieren. Dies könnte man mit `malloc()` etc. bewerkstelligen. Allerdings hat man dann wieder das Problem, dass `malloc()` unter

Umständen keinen Speicherplatz mehr bekommt, was immer eine entsprechende Fehlerbehandlung nach sich zieht. Eine andere Möglichkeit wäre, die Tabellenstruktur so abzuändern, dass alle Funktionsnamen in einem gesonderten Array liegen. Aber das würde langfristig die Flexibilität der internen Realisierung deutlich einschränken.

Ein erster Iterator: Obiges Ziel kann man auch mittels eines *Iterators* erreichen, der in Anlehnung an die Programmiersprache Sather [14] auch `iter()` genannt wird. Ein derartiger Iterator ist ein endlicher Automat, der ein Gedächtnis benötig. Dieses Gedächtnis kann man beispielsweise mit einem Zeiger vom Typ `void *` erreichen, den man als Parameter übergibt und in dem der Iterator Informationen ablegen kann, die er für zukünftige Aufrufe benötigt. Eine mögliche Realisierung nebst eines Testprogramms sieht wie folgt aus (die Funktion `ic_iter()` muss natürlich in der Datei `icalc.c` definiert sein).

```
1  // the following function goes to icalc.c/h
2  char *ic_iter( void **iter )
3       {
4          ICP ip = !*iter? ftab: *iter;
5          *iter = ip < ftab + TAB_SIZE? ip + 1: 0;
6          return *iter? ip->ic_name: 0;
7       }
8
9  //this is the test program and goes somewhere else
10 int main( int argc, char **argv )
11     {
12        void *iter = 0;
13        char *cp;
14        printf( "available icalc functions are:" );
15        while( cp = ic_iter( & iter ) )
16           printf( " %s", cp );
17        printf( "\n" );
18     }
```

Das Testprogramm gibt folgendes aus: `available icalc functions are: + - * /`

Das Verstehen einer derartigen Implementierung benötigt etwas Zeit. Am besten geht dies wie üblich mit einer kleinen Handsimulation. Der wesentliche Grundgedanke ist, dass anfangs das Gedächnis der Funktion, der Zeiger `iter` auf `0` gesetzt wird. Ein Nebeneffekt dieses Ansatzes ist folgender: Schreibt man die Zeilen 14 bis 17 ein zweites Mal hin, würde die Liste ein zweites Mal ausgegeben werden, da am Ende der ersten Liste der Zeiger `iter` wieder auf `0` gesetzt wurde.

Eine alternative Implementierung: Obige Implementierung des Iterators kann man noch etwas komprimieren, in dem man den zurückgegebenen Zeiger auf den Funktionsnamen als Gedächtnis verwendet. Der folgenden Implementierung liegt der recht einfache Gedanke zugrunde, dass zwei Namenszeiger immer den Abstand `sizeof( ICALC )` haben müssen, egal wo konkret sie innerhalb der Struktur `ICALC` plaziert werden. Die Implementierung nebst Testprogramm sieht wie folgt aus:

```
1  // the following function goes to icalc.c/h
2  char **ic_iter( void *iter )
3       {
4          iter = iter? iter + sizeof( ICALC ): & ftab->ic_name;
5          return ((ICP)iter >= ftab + TAB_SIZE)? 0: iter;
6       }
7
8  //this is the test program and goes somewhere else
9  int main( int argc, char **argv )
10      {
11         char **cp = 0;
12         printf( "available icalc functions are:" );
13         while( cp = ic_iter( cp ) )
14            printf( " %s", *cp );
15         printf( "\n" );
16      }
```

Auch hier gilt wieder, dass der Namenszeiger (also das Gedächtnis des Iterators) vor dem ersten und nach dem letzten Aufruf des Iterators den Wert `0` hat.

Ein `int`-basierter Iterator: Der eine oder andere mag argumentieren, dass als Gedächtnis eine einfache `int` Zahl genügt hätte. Doch dann würde man sich sehr stark auf die Organisation mittels einer internen Tabelle fixieren; eine interne Organisation mittels Listen etc. würde die Implementierung des Iterators stark erschweren. Ferner ist zu bedenken, dass ein `int` häufig kleiner als ein Zeiger ist, sodass man für das innere Gedächtnis auch keine Aufrufe von `malloc()` verwenden kann. Aus diesen Gründen erscheinen obige Implementierungen sinnvoller. Trotz aller Bedenken, eine auf `int`-Zahlen basierende Implementierung könnte wie folgt aussehen:

```
1  // the following function goes to icalc.c/h
2  char *ic_iter( int *iter )
3       {
4          int ind = *iter;
5          *iter = ind < TAB_SIZE? ind + 1: 0;
6          return ind < TAB_SIZE? (ftab + ind)->ic_name: 0;
7       }
```

Kapitel 87

Opaque Datentypen

„Schon wieder so etwas komisches von Euch! Ich weiß nicht einmal, wie man das Wort Opaque überhaupt ausspricht“ Die Aussprache ist wie „Opal“, nur mit einem „k“ am Ende. Dahinter verbirgt sich, dass die konkrete Ausgestaltung eines (abstrakten) Datentyps nach außen verborgen bleibt; sie ist nur innerhalb des Implementierungsmoduls bekannt. In diesem Kapitel erklären wir das Konzept des opaquen Datentyps am Beispiel eines Stacks.

87.1 Einführung am Beispiel eines Stacks

Im Grunde genommen hatten wir schon in Kapitel 74 eine Art opaquen Datentyp kennengelernt, denn der dort entwickelte Stack erfüllte alle oben genannten Anforderungen hinsichtlich Kapselung. Diese Kapselung gelang uns aufgrund folgender Annahme: Wir benötigen maximal einen Stack pro C-Programm. Dadurch konnten wir unser Problem mit einem dateiglobalen Zeiger auf den Anfang des Stacks lösen. Dadurch benötigten alle Zugriffsfunktionen nur einen Zeiger auf die Daten, die auf den Stack abgelegt werden sollen.

So schön wie die in Kapitel 74 präsentierte Lösung auch sein mag, so hat sie doch zumindest die folgenden Einschränkungen:

1. Die Implementierung des Stacks muss wissen, wie der zu speichernde Datentyp aussieht. Dadurch müssen wir jedesmal den Stack anpassen bzw. neu übersetzen, wenn sich hier etwas ändert. Aber zur Beseitigung dieser Einschränkung kommen wir erst in Kapitel 88.
2. Mit der gewählten Realisierung können wir nur einen Stack pro C-Programm verwalten; weder können wir mehr als einen vom selben Typ noch welche eines anderen Typs gleichzeitig verwenden.

Für fortgeschrittene Programmierer ist insbesondere der zweite Punkt ein echt gravierender Nachteil. Im Folgenden denken wir laut über mögliche Alternativen nach:

Möglichkeit 1: Vervielfältigen des Moduls: Wir könnten für jeden neuen Stack das erstellte Modul duplizieren. Allerdings müssten wir zusätzlich alle Namen ändern, damit es später beim Linken keine Probleme gibt. Aufgrund des stark aufgeblähten Quelltextes verfolgen wir diese Möglichkeit nicht weiter.

Möglichkeit 2: Explizite Stack-Zeiger:
Wir könnten alle Stack-Funktionen um einen Zeiger erweitern, der auf den Anfang des Stacks zeigt. In diesem Falle müsste man aber den Datentyp `STACK`, den wir bisher in der Modulimplementierung (der `.c`-Datei) verbergen konnten, in die Header-Datei packen und somit nach außen bekannt machen. Dies würde funktionieren, wollen wir aber nicht; wir wollen ja gerade den Verwaltungsteil des Stacks in der Modulimplementierung verbergen.

Möglichkeit 3: Stack Array:
Im Stack Modul könnten wir ein Array `static SP stacks[ SIZE ]` von Stack-Zeigern definieren. Dazu bräuchten wir noch eine Variable `static int nextStack = -1`, über die wir den nächsten freien Index bekommen. Ein Aufruf einer neuen Funktion `new_stack()` wurde uns den nächsten freien Index liefern und die dateiglobale Variable `nextStack` um eins erhöhen, sofern im Array noch Platz ist. Wie wir ggf. dieses Array vergrößern, haben wir in Kapitel 70 kennengelernt. Den von der Funktion `new_stack()` zurückgegebenen Index würde man als *Handle* bezeichnen, den man bei allen weiteren Stack-Funktionen als ersten Parameter übergeben müsste. Eine derartige Handle-basierter Realisierung ist durchaus üblich und für unsere Anwendung völlig ausreichend. Aber so richtig überzeugend ist diese Variante auch nicht. Zumindest legt ein derartiger Handle-basierte Ansatz eine Array-basierende Implementierung nahe; andere Varianten wie Listen etc. sind eher mühsam zu realisieren.

Möglichkeit 4: Opaquer Datentyp:
Diese recht elegante Möglichkeit diskutieren wir im nächsten Abschnitt.

87.2 Opaquer Datentyp mittels `void`-Zeiger

Die Realisierung eines opaquen Datentyps ist eigentlich sehr einfach, wir müssen nur die `.h`-Dateien ein wenig anders schreiben als die `.c`-Dateien. Das heißt, wir würden in der `.h`-Datei den Datentyp Stack als einfachen `void`-Zeiger definieren. Ein derartiger Zeiger würde als erster Parameter an alle Funktionen übergeben werden, die *intern* diesen Zeiger nicht als `void`-Zeiger sondern mittels Casts als `STACK`-Zeiger behandeln. Zur näheren Erläuterung hier einfach mal ein paar Code-Schnipsel (fehlende Funktionen müssen aus obigen Quelltexten ergänzt werden):

stack.h

```
 1 /*---------- stack.h ---------- */
 2 void *new_stack(),         *popEntry( void *sp );
 3 int   isEmpty( void *sp ),  pushEntry( void **sp, DP dp );
 4 DP    getFirst( void *sp );
```

stack.c

```
 1 /*---------- stack.c ---------- */
 2
 3 #include <stdlib.h>
 4 #include "data.h"
 5 #include "stack.h"
 6
 7 typedef struct _stack {
 8                 struct _stack *next;
 9                 DATA data;
10         } STACK, *SP;
11
12 void *new_stack()
13      { return 0; }
14
15 int pushEntry( void **sp, DP dp )
16     {
17        SP p = mk_element( dp, (SP) sp );
18        if ( p != 0 )
19           *sp = p;
20        return p != 0;
21     }
22
23 void *popEntry( void *sp )
24      {
25         SP p = sp;
26         if ( ! isEmpty( sp ) )
27         { sp = sp->next; free( p ); }
28         return sp;
29      }
```

Obige Implementierung sollte eigentlich klar sein. Ein zusätzlicher Hinweis betrifft die Funktion `new_stack()`. In der momentanen Implementierung gibt diese Funktion nur einen Null-Zeiger zurück. Aber prinzipiell könnten hier auch andere Anweisungen stehen. Ein möglicher Anwendungsfall betrifft das Ablegen von Zusatzinformationen. Eine andere Möglichkeit betrifft einen weiteren Zeiger, um die Implementierung zu vereinfachen, wie wir im nächsten Abschnitt besprechen.

87.3 Vereinfachter Opaquer Stack

Die im vorherigen Abschnitt vorgestellte Implementierung ist schon ganz ok. Das einzig Unschöne ist, dass sich der von `new_stack()` zurückgegebene Zeiger bei jeder Push- und Pop-Operation verändert; das ist nicht weiter schlimm, macht aber die Funktionsdeklarationen und -implementierungen etwas umständlich. Die folgende Implementierung beseitigt diese kleine Schwäche, indem die Funktion `new_stack()` keinen Null-Zeiger zurückgibt, sondern ein Dummy-Element produziert, an das der eigentliche Stack angehängt wird.

```
1  #include <stdlib.h>
2  #include "data.h"
3  #include "stack.h"
4
5  typedef struct _stack {
6                  struct _stack *next;
7                  DATA data;
8          } STACK, *SP;
9
10 static SP mk_element( DP dp, SP next ){
11           SP p = malloc( sizeof( STACK ) );
12           if ( p != 0 )
13           { p->next = next; p->data = *dp; }
14           return p;
15       }
16
17 void *new_stack()
18      { DATA dummy; return mk_element( & dummy, 0 ); }
19
20 int isEmpty( void *sp )
21     { return ((SP) sp)->next == 0; }
22
23 DP  getFirst( void *sp )
24     { return isEmpty( sp )? 0: & ((SP) sp)->next->data; }
25
26 int pushEntry( void *sp, DP dp ){
27        SP p = mk_element( dp, ((SP) sp)->next );
28        ((SP) sp)->next = p? p: ((SP) sp)->next;
29        return p != 0;
30     }
31 int popEntry( void *sp ){
32        SP stack = sp, p = stack->next;
33        if ( ! isEmpty( sp ) )
34        { stack->next = p->next; free( p ); }
35        return 1;
36     }
```

Kapitel 88

Generische Datentypen

Im vorherigen Kapitel haben wir opaque Datentypen kennengelernt, die die Details eines abstraken Datentyps recht gut kapseln. Zusätzlich haben wir besprochen, wie wir auf bequeme Art und Weise gleich mehrere „Instanzen“ eines derartigen opaquen Datentyps verwenden können. Nun haben wir aber immer noch das Problem, dass wir nur einen Typ von Nutzdaten mit solch einem abstrakten Datentyp, in unserem Fall ein Stack, verwalten können. Wollen wir in einer Anwendung einen weiteren Stack, aber mit anderen Daten verwalten, müssen wir alle Algorithmen duplizieren und die Namen aller Funktionen modizieren. Das ist aber weder ökonomisch noch änderungsfreundlich. Bemerken wir beispielsweise einen Fehler in einer der Funktionen, müssen wir die Änderungen in allen Varianten nachtragen. In diesem Kapitel besprechen wir, wie wir die Algorithmen mittels generischer Datentypen nur einmal entwickeln müssen.

88.1 Problemstellung und Lösungsansatz

Schauen wir nochmals zurück zu den vorherigen Implementierungen eines Stacks. So ein Stack (wie auch andere dynamische Datenstrukturen) besteht im Grunde genommen aus zwei Teilen, einem Datenteil und einem Verwaltungsteil, der bei einem Stack lediglich aus einem Zeiger zum nächsten Element besteht. Die Problematik, dass wir die Administrationsstruktur verbergen wollen, haben wir im vorherigen Kapitel durch das Konzept des opaquen Datentyps gelöst. Bleibt noch der Datenteil. Um die Daten vom Stack verwalten zu können, mussten wir den entsprechenden Datentyp in die Stack-Implementierung einbinden und alle Zugriffsfunktionen entsprechend typgerecht deklarieren und implementieren. So lernt man es, so funktioniert es und so schränkt man eine Stack-Implementierung auf genau einen Datentyp ein.

Bei längerem Überlegen stellt sich die Frage, ob die Stack-Implementierung überhaupt die Daten kennen muss. Eigentlich nicht, denn der Stack schaut sich die Daten nie an. Einzige Ausnahme von dieser Aussage: Der Stack muss in unseren bisherigen Implementierungen

immer wissen, wie viele Bytes durch die Daten belegt sind, damit er ein neues Element alloziieren kann (`mk_element()`). Aber dieser letzten Abhängigkeit könnten wir uns einfach entledigen, in dem wir die Daten nicht in die Stack-Datenstruktur integrieren sondern mittels eines `void`-Zeigers referenzieren. Die folgenden beiden Code-Schnipsel stellen beide Varianten einander gegenüber:

```
1 typedef struct _stack {
2         struct _stack *next;
3         DATA data;
4 } STACK, *SP;
```

```
1 typedef struct _stack {
2         struct _stack *next;
3         void *dp;
4 } STACK, *SP;
```

Wie einfach zu sehen ist, hat die rechte Seite keinen konkreten Bezug mehr zum eigentlichen Datentyp der „Nutzdaten“. Allerdings sollte man sich darüber im klaren sein, dass man sich diese Generalisierung durch eine etwas reduzierte Typüberprüfung erkauft.

88.2 Beispiel: Generischer Stack I

Mit obiger kleinen Änderung ist die Realisierung eines generellen Stacks sehr einfach. Wir brauchen eine `.h`-Datei:

```
1  /*
2   * module:      gstack.h
3   *
4   * description: interface of a general stack
5   *
6   */
7
8  void *new_stack();
9  int   isEmpty( void *stack );
10 void *getFirst( void *stack );
11 int   pushEntry( void **stack, void *dp );
12 void *popEntry( void *stack, void **dp );
```

Wie leicht zu sehen ist, baut diese Realisierung auf das Beispiel von Abschnitt 87.2 auf. Wie im vorherigen Kapitel auch, werden wir dieses Beispiel im nächsten Abschnitt noch weiterentwickeln.

Auf der nächsten Seite sehen wir die Realisierung, die aufgrund der bisherigen Diskussion keine weiteren Schwierigkeiten beinhalten sollte.

```
1  /*
2   * module:      gstack.c
3   *
4   * description: implementation of a general stack
5   *
6   */
7
8  #include <stdlib.h>
9  #include "gstack.h"
10
11 typedef struct _stack {
12                 struct _stack *next;
13                 void *dp;
14         } STACK, *SP;
15
16 void *new_stack()
17      { return 0; }
18
19 int isEmpty( void *stack )
20     { return stack == 0; }
21
22 void *getFirst( void *stack )
23      { return stack? ((SP) stack)->dp: 0; }
24
25 int pushEntry( void **stack, void *dp )
26     {
27        SP p = malloc( sizeof( STACK ) );
28        if ( ! p )
29           return 0;
30        p->next = *stack; p->dp = dp; *stack = p;
31        return 1;
32     }
33
34 void *popEntry( void *stack, void **dp )
35     {
36        void *p = stack? ((SP) stack)->next: 0;
37        if ( dp )
38           *dp = stack? ((SP) stack)->dp: 0;
39        free( stack );
40        return p;
41     }
```

Zur Veranschaulichung haben wir auch eben ein kleines Beispielprogramm entwickelt, das diesen Stack verwendet. Der Quelltext sieht wie folgt aus:

```
1  #include <stdio.h>
2  #include "gstack.h"
3
4  typedef struct {              // zunaechst unsere alten Daten
5                  int  ivalue;  // der eine Wert
6                  char cvalue;  // der andere Wert
7          } DATA, *DP;          // Typ: Daten & Pointer
8
9  void prt_data( DP dp )
10      { printf( "\ti= %2d c= '%c'\n",dp->ivalue,dp->cvalue ); }
11
12 int main( int argc, char **argv )
13     {
14        DATA d1 = { 1, 'a' }, d2 = { 2, 'b' }, d3 = { 3, 'c' };
15        void *stack = new_stack( sizeof( DATA ) );
16        if (    pushEntry(&stack,&d1) && pushEntry(&stack,&d2)
17             && pushEntry(&stack,&d3) && pushEntry(&stack,&d1))
18           for( ; ! isEmpty(stack); stack = popEntry(stack, 0))
19              prt_data( getFirst( stack ) );
20        else fprintf( stderr, "Sorry, Speicher zu knapp\n" );
21        return 0;
22     }
```

An diesem Beispiel kann man nochmals sehr gut sehen, dass es typmäßig keine Abhängigkeiten mehr zwischen der Stack Implementierung und dem Datentyp der Nutzerdaten besteht. Für die meisten sollte klar sein, wie man jetzt gleichzeitig zwei Stacks verwalten kann. Zur Veranschaulichung aber dennoch hier folgendes Beispiel:

```
1  #include <stdio.h>
2  #include "gstack.h"
3
4  int main( int argc, char **argv )
5      {
6        double d1 = 1.0, d2 = 2.0;
7        int    i1 = -1,  i2 = -2;
8        void  *s1 = new_stack(), *s2 = new_stack();
9        pushEntry( & s1, & d1) && pushEntry( & s1, & d2);
10       pushEntry( & s2, & i1) && pushEntry( & s2, & i2);
11       for( printf( "s1:" ); !isEmpty(s1); s1=popEntry(s1,0))
12          printf( " %e", * (double *) getFirst( s1 ) );
13       printf( "\n" );
14       for( printf( "s2:" ); !isEmpty(s2); s2=popEntry(s2,0))
15          printf( " %d", * (int *) getFirst( s2 ) );
16       printf( "\n" );
17     }
```

88.3 Beispiel: Generischer Stack II

Der generische Stack aus dem vorherigen Abschnitt ist schon sehr gut. Doch es wäre schön, wenn der Stack auf Wunsch auch gleich noch eine Kopie der abzulegenden Daten erstellen würde. In diesem Abschnitt werden wir genau das machen, und entsprechend zwei Funktionen zum Initialisieren des Stacks anbieten. Wir werden die folgenden beiden Fälle unterscheiden:

1. Die einfachste Variante erzeugt keine Kopien, sondern legt den übergebenen Datenzeiger direkt auf dem Stack ab. Dies können einerseits statische Objekte sein oder welche, die vorher (vom Aufrufer) mittels `malloc()` auf dem Heap eingerichtet wurden.
2. Die zweite Variante erzeugt für jeden abzulegenden Datensatz eine Kopie auf dem Heap und kopiert die Nutzerdaten (`*dp`) dort hinein. Für diese Funktionalität „merkt“ sich der Stack beim Initialisieren die Größe des abzulegenden Datensatzes.

Wie im zweiten Beispiel des vorherigen Kapitels erzeugen wir einen „Kopfdatensatz“, der den aktuellen Stack-Typ sowie die Größe des Datensatzes sowie den Zeiger auf eine Funktion zum Duplizieren des Datensatzes speichert. Und da wir schon einen ganzen „Kopfdatensatz“ benötigen, können wir gleich noch Platz für einen Iterationszeiger vorsehen. Die `.h`-Datei für diesen generalisierten, opaquen Stack sieht wie folgt aus:

```
 1 /*
 2  * module:      gs-stack.h
 3  *
 4  * description: implementation of a general stack that
 5  *              is able to save the data on the heap.
 6  */
 7
 8 void *new_simple_stack();
 9 void *new_save_stack( int dsize );
10
11 int   isEmpty( void *stack );
12 void *getFirst( void *stack );
13
14 int   pushEntry( void *stack, void *dp );
15 void *popEntry( void *stack );
16
17 void *iter( void *stack );
```

Die Realisierung dieses erweiterten, generalisierten Stacks sollte keine größeren Schwierigkeiten bereiten. Sie ist natürlich so angelegt, dass gleich mehrere Stacks verwaltet werden können. Der Quelltext sieht wie folgt aus:

```
1  /*
2   * module:      gs-stack.c
3   *
4   * description: implementation of a general stack that
5   *              is able to save the data on the heap.
6   */
7
8
9  #include <stdlib.h>
10 #include "gs-stack.h"
11
12 typedef struct _element {
13                 struct _element *next;
14                 void *dp;
15         } ELEMENT, *EP;
16
17 typedef struct _stack {
18                 EP   stack, iter;// the stack data and an iter
19                 int  size;       // data size; for save stack
20                 void *(*mk_new)(void *p, int size); // for new
21         } HEAD, *HP;
22
23 static void *dp_nosave( void *dp, int size )
24        { return dp; }
25
26 static void *dp_save( void *dp, int size )
27        {
28           char *p, *to, *from = dp;
29           if ( p = to = malloc( size ))
30              while( size-- )
31                 *to++ = *from++;
32           return p;
33        }
34
35 static void *mk_hdr( int save, int size )
36        {
37           HP p = malloc( sizeof( HEAD ) );
38           if ( p )
39           {
40              p->stack = 0; p->iter = 0; p->size = size;
41              p->mk_new = (save)? dp_save: dp_nosave;
42           }
43           return p;
44        }
```

```
45 void *new_simple_stack()
46      { return mk_hdr( 0, 0 ); }
47
48 void *new_save_stack( int size )
49      { return mk_hdr( 1, size ); }
50
51 int isEmpty( void *stack )
52     { return !stack || ((HP) stack)->stack == 0; }
53
54 void *getFirst( void *sp )
55      { return sp && ((HP) sp)->stack?((HP)sp)->stack->dp:0; }
56
57 int pushEntry( void *stack, void *dp )
58     {
59        HP hp = stack;
60        EP p = malloc( sizeof( ELEMENT ) );
61        if ( ! p || ! (p->dp = (* hp->mk_new)( dp, hp->size )))
62           { free( p ); return 0; }
63        p->next = hp->stack; hp->stack = p;
64        return 1;
65     }
66
67 void *popEntry( void *stack )
68      {
69         HP hp = stack;
70         EP ep = (hp && hp->stack)? hp->stack: 0;
71         void *dp = 0;
72         if ( ep )
73         {
74            hp->iter == ep || (hp->iter = ep->next);
75            dp = ep->dp; hp->stack = ep->next; free( ep );
76         }
77         return dp;
78      }
79
80 void *iter( void *stack )
81      {
82         HP hp = stack;
83         hp->iter = hp->iter? hp->iter->next: hp->stack;
84         return hp->iter? hp->iter->dp: 0;
85      }
```

Das Beispielprogramm auf der folgenden Seite zeigt, wie dieses Stack-Modul verwenden werden kann:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "gs-stack.h"
4
5  typedef struct {
6                  int  ival;
7                  char cval;
8          } DATA, *DP;
9
10 int prt_data( DP p )
11     { return p &&
12               printf( "\ti= %2d c= '%c'\n",p->ival,p->cval ); }
13
14 int main( int argc, char **argv )
15     {
16        DATA d1 = { 1, 'a' }, d2 = { 2, 'b' }, d3 = { 3, 'c' };
17        int i1 = 815, i2 = 4711;
18        void *s1, *s2;
19        if ( (s1 = new_save_stack( sizeof( DATA )))
20             && pushEntry( s1, & d1 ) && pushEntry( s1, & d2 )
21             && pushEntry( s1, & d3 ) && pushEntry( s1, & d1 ))
22           while( prt_data( iter( s1 ) ) )
23              ;
24        else printf( "Sorry, a pushEntry(s1) has failed\n" );
25        if ((s2 = new_save_stack( sizeof( int )))
26             && pushEntry( s2 , & i1 ) && pushEntry(s2, & i2))
27           for( ; ! isEmpty( s2 ); popEntry( s2 ) )
28              printf( "  i= %d\n", * (int *) getFirst( s2 ) );
29        else  printf( "Sorry, a pushEntry(s2) has failed\n" );
30     }
```

In diesem (recht kompakten) Testprogramm wird der erste Stack `s1` mittels des Iterators und der zweite Stack `s2` über das Abräumen ausgegeben.

88.4 Ergänzungen

Das Konzept der generischen Datentypen mittels `void`-Zeigern kann recht universell eingesetzt werden. Beispielsweise könnte der Taschenrechner aus Kapitel 85 in ähnlicher Weise erweitert werden. Statt die Operanden direkt über ihre Werte zu übergeben, könnte man dies auch mittels `void`-Zeigern realisieren. Die einzelnen Berechnungsfunktionen müssten die Zeiger mittels eines Cast entsprechend umwandeln. Dadurch könnte man durch weitere Berechnungsfunktionen auch andere Datentypen wie beispielsweise `double` verwenden. Ein entsprechender Versuch wäre eine gute Übung für jeden Leser.

Kapitel 89

Erhöhte Sicherheit bei generischen Datentypen

Durch die Verwendung generischer Datentypen mittels `void`-Zeigern brauchen wir einen Algorithmus unabhängig von den verwendeten Datentypen nur einmal implementieren. Das führt zu einer deutlichen Kürzung des Quellcodes und einer deutlich verbesserten Änder- und Wartbarkeit.

Problembeschreibung: Die Verwendung generischer Datentypen bezahlen wir mit dem teilweisen Ausschalten der Typüberprüfung, was die Fehleranfälligkeit erhöht. Es könnte passieren, dass wir statt eines Zeigers auf einen Stack einen Zeiger auf eine Zeichenkette übergeben. Bei der eher konventionellen Methode hätten wir zumindest die Chance, dass der Compiler etwas merkt, bei der Methode der letzten Abschnitte hätten wir keine. Im Folgenden zeigen wir eine Methode, mittels derer man zumindest die Chance auf Fehlererkennung (nicht -vermeidung) in einfacher Weise deutlich verbessern kann.

Lösungsansatz: Der folgende Lösungsansatz ist verblüffend einfach. Im Implementierungsmodul eines generischen Datentyps können wir uns eine dateiglobale Variable definieren. Bei jeder neuen Instanz (jedem neuen Stack in obigen Beispielen) setzen wir einen Zeiger auf diese Variable, den wir in allen Zugriffsoperationen überprüfen. Ein entsprechender Code-Schnipsel befindet sich auf der nächsten Seite. Eine Übertragung auf andere generischen Datentypen sollte sehr einfach sein.

```
1  typedef struct _stack {
2                  int  *magic;    // pointer to a file
3                                  // global variable
4                  .......
5          } HEAD, *HP;
6
7  static int magic;
8
9  static void *mk_hdr( int save, int size )
10        {
11           HP p = malloc( sizeof( HEAD ) );
12           if ( p )
13           {
14              p->magic = & magic;
15              .....
16           }
17           return p;
18        }
19
20 static void enforce( HP p )
21        {
22           if ( ! p || p->magic != & magic )
23           {
24              fprintf( stderr,
25                       "secure stack: invalid pointer %p\n",
26                       p );
27              exit( p? 2: 1 );
28           }
29        }
30
31 int pushEntry( void *stack, void *dp )
32     {
33        HP hp = stack;
34        EP p = malloc( sizeof( ELEMENT ) );
35        enforce( hp );
36        ....
37     }
```

Kapitel 90

Weitere Funktionalitäten des Präprozessors

Den Präprozessor haben wir eigentlich schon in Kapitel 38 behandelt. Dabei haben wir uns aber auf diejenigen Funktionalitäten beschränkt, die für Anfänger und leicht Fortgeschrittene wesentlich sind. Hier nun präsentieren wir noch ein paar Dinge, die recht nützlich sein können.

90.1 Erstellen von Zeichenketten

Mittels der `#define`-Direktive können wir „Labels“ und Makros definieren, die im weiteren Quelltext ersetzt werden. Allerdings werden diese Ersetzungen nicht in konstanten Zeichenketten durchgeführt. Manchmal wäre das aber wünschenswert, sodass wir es hier zeigen. Der wesentliche Ansatzpunkt ist das Doppelkreuz `#` auf der rechten Seite einer `#define`-Direktive. Ein einfaches Beispiel sieht wie folgt aus:

Quelltext

```
1 #define quote(x)          #x
2
3 printf( quote(Hello world \n) )
```

Nach `cpp -P datei.c`

```
1
2
3 printf( "Hello world \n" )
```

Das Makro in Zeile 1 besagt, dass das Argument `x` in Gänsefüßchen eingeschlossen wird, wie man auf der rechten Seite in Zeile 3 sehen kann.

Nun gibt es noch eine Schwierigkeit: das Argument `x` wird so genommen wie es ist. Mit anderen Worten: das Argument wird nicht noch einmal evaluiert, was zu einem unerwünschten Ergebnis führen kann. Hier ein Beispiel mit einem vermutlich unerwünschten Ergebnis:

Quelltext

```
1 #define quote(x)        #x
2 #define NAME    Inge Mustermann
3
4 printf( quote( NAME ) )
```

Nach `cpp -P datei.c`

```
1
2
3
4 printf( "NAME" )
```

Wie man gut sehen kann, wird das Argument `NAME` nicht durch `Inge Mustermann` ersetzt, sondern so genommen, wie es ist. Ein erneutes Evaluieren erreicht man durch einen verschachtelten Aufruf:

Quelltext

```
1 #define _quote(x)         #x
2 #define quote(x)   _quote(x)
3 #define NAME      Peter Frau
4
5 printf( quote( NAME ) )
6 printf( quote( Hallo NAME ) )
```

Nach `cpp -P datei.c`

```
1
2
3
4
5 printf( "Peter Frau" )
6 printf( "Hallo Peter Frau" )
```

Wie man in Zeile 5 sehen kann, wird nun auch das „Label“ `NAME` durch `Peter Frau` ersetzt und in eine Zeichenkette umgewandelt.

Zeile 6 zeigt noch ein weiteres Beispiel, in dem das erste Wort unverändert bleibt. Das folgende Programmstück zeigt ein durchaus sinnvolles Beispiel:

```
 1 #include <stdio.h>
 2
 3 #define _quote( x )     #x
 4 #define quote( x )      _quote(x)
 5
 6 #define texp( x )       "exp= '%s' val= %d\n", quote( x ), x
 7
 8 int main( int argc, char **argv )
 9     {
10        int a = 2, b = 1, c = 0;
11        printf( texp( a > b && c < 3 ) );
12     }
```

Das Makro `texp( x )` in Zeile 6 erzeugt drei Teile, eine Formatierung, einen beliebigen in eine Zeichenkette umgewandelten Ausdruck sowie den Ausdruck selbst. Dieses Programm erzeugt folgende Ausgabe: `exp= 'a > b && c < 3' val= 1`.

Dieser `#`-Operator lässt sich in vielen Anwendungen sehr schön einsetzen. Beispielsweise kann man damit seine Webseiten konsistent halten.

90.2 Variable Anzahl von Argumenten

Wenn man kurz überlegt, wird man zu dem Schluss kommen, dass man kein Komma als Argument übergeben kann. Daher kann folgendes Programm nicht vom Präprozessor korrekt verarbeitet werden:

```
1 #define errmsg( fp, msg )        fprintf( fp, msg )
2
3 errmsg(stderr,"Datei %s kann nicht geoeffnet werden\n",x.txt);
```

In unserem Beispiel wäre die gewünschte Substitution wie folgt gewesen:
`fprintf( stderr, "Datei %s kann nicht geoeffnet werden\n", x.txt )`.

Für solche Anwendungsfälle gibt es die Möglichkeit einer variablen Anzahl von Argumenten: Diese werden auf der linken Seite mittels dreier Punkte `...` und auf der rechten Seite mittels `__VA_ARGS__` notiert. Folgende Beispiel sollte dies klären:

```
1 #define errmsg( fp, ... )        fprintf( fp,__VA_ARGS__ )
2
3 errmsg(stderr,"Datei %s kann nicht geoeffnet werden\n",x.txt);
```

Jetzt kommt es zur gewünschten Ersetzung:

```
1 fprintf( stderr, "Datei %s kann nicht geoeffnet werden\n",
2          x.txt );
```

Abschließend noch ein weiteres Beispiel:

```
1 #define friends( ... )           all my friends: __VA_ARGS__
2
3 friends( peter, paul and marie )
```

Der Aufruf von `cpp -P datei.c` führt zu `all my friends: peter, paul and marie`.

90.3 Komposition von „Labels“

Als letztes stellen wir ein kleines Feature vor, das vor allem die Wartbarkeit und Änderbarkeit von Programmen erhöht. Nehmen wir an, wir hätten ein Makro, das uns eine Art Visitenkarte von Personen zusammenstellt. Bei den Daten könnte es sich um den Namen und die Telefonnummer handeln. Nehmen wir weiter an, wir hätten etwa 100 Personen in unserer Liste. Sollten wir nun auf die Idee kommen, eine weitere Angabe wie beispielsweise die E-Mail-Adresse zu benötigen, müssten wir alle 100 Aufrufe dieses Makros anpassen. Das wäre sehr mühsam, insbesondere wenn die Personen über mehrere Dateien verteilt sind. Auch für dieses Problem bietet der Präprozessor eine Lösung an. Diese besteht darin,

dass man auf der rechten Seite eines Makros mittels `##` ein „Label“ zusammenbauen kann. Das Einfachste ist, das ganze anhand eines Beispiels zu erklären:

```
 1 #define Peter_Name      Peter Mann
 2 #define Peter_Phone     01188
 3 #define Peter_Email     no-plan@nirwana.mars
 4
 5 #define Lara_Name       Lara Frau
 6 #define Lara_Phone      01188
 7 #define Lara_Email      also-no-plan@nirwana.mars
 8
 9 #define bcard( person ) \
10                 Name: person ##_Name \
11                 Phone: person ##_Phone \
12                 Email: person ##_Email
13
14 bcard( Peter )
15 bcard( Lara )
```

Obiges Beispiel definiert zwei Personen mit je drei Angaben. Wie man sehen kann, werden in den Zeilen 14 und 15 nur die beiden Namen `Peter` und `Lara` übergeben. Aber auf der rechten Seite des Makros `bcard()` (Zeilen 9 bis 12) werden diese Namen zu den vollständigen „Labels“ ergänzt. Nach Aufruf des Präprozessors ergeben sich folgende Ersetzungen:

```
1 Name: Peter Mann Phone: 01188 Email: no-plan@nirwana.mars
2 Name: Lara Frau Phone: 01188 Email: also-no-plan@nirwana.mars
```

Teil X

Anhänge

Anhang A

ASCII-Tabelle

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1...	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2...	*SP*	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*DEL*

Anhang B

Präzedenztabelle

Operator	Beschreibung	Assoziativität
`()`	runde Klammern (Gruppierung)	von links nach rechts
`[]`	eckige Klammern (Array Indizes)	
`.`	Komponentenzugriff über Variablenname	
`->`	Komponentenzugriff über Zeiger	
`++ --`	Post Inkrement/Dekrement	
`++ --`	Pre Inkrement/Dekrement	von rechts nach links
`+ -`	Vorzeichen	
`! ~`	Logisch Negation/1er Komplement	
`(type)`	Cast	
`*`	Zeiger-Dereferenzierung	
`&`	Adressoperator	
`sizeof`	Größe in Bytes	
`* / %`	Multiplikation/Division/Modulo	von links nach rechts
`+ -`	Addition/Subtraktion	von links nach rechts
`<< >>`	Bit-Shift links/rechts	von links nach rechts
`< <=`	Relation Kleiner/Kleiner-Gleich	von links nach rechts
`> >=`	Relation Größer/Größer-Gleich	
`== !=`	Relation Gleich/Ungleich	von links nach rechts
`&`	Bitweises „UND“	von links nach rechts
`^`	Bitweises „XOR“	von links nach rechts
`|`	Bitweises „ODER“	von links nach rechts
`&&`	Logisches „UND“	von links nach rechts
`||`	Logisches „ODER“	von links nach rechts
`?:`	Dreiwertige Bedingung	von links nach rechts
`=`	Zuweisung	von rechts nach links
`+= -=`	Addition/Subtraktion-Zuweisung	
`*= /=`	Multiplikation/Division-Zuweisung	
`%= &=`	Modulo/bitweises „UND“ Zuweisung	
`^= |=`	Bitweises „XOR“/„ODER“ Zuweisung	
`<<= >>=`	Bit-Shift links/rechts Zuweisung	
`,`	Komma, Ausdrucks-Liste	von links nach rechts

Anhang C

Kurzfassung der Ausgabeformatierungen

Dieser Anhang fasst alle im Skript an verschiedenen Stellen erwähnten Ausgabeformatierungen zusammen, da diese auch bei vielen fortgeschrittenen Programmierern immer wieder zu Verwirrungen führen. Allerdings führen wir hier nur einige der wichtigsten Formatierungen auf; für einen vollständigen Überblick ist die Dokumentation heranzuziehen. Unter Linux geht dies beispielsweise mit dem Kommando: `man 3 printf`

Regeln: Die generellen Regeln der Ausgabeformatierung lauten wie folgt:

1. Formatierungen werden immer mit einem `%` eingeleitet, das von einer konkreten Formatierung gefolgt wird.
2. Will man ein %-Zeichen ausgeben, so muss ein zweites `%`-Zeichen folgen.

   Beispiel: `printf( "%%");` gibt ein einzelnes Prozentzeichen aus.
3. Die „üblichen“ Formatierungen sind: `c, s, d, f, e, x, p` usw.
4. Die meisten Formatierungen können durch eine oder zwei „Längenangaben“ ergänzt werden.

   Beispiel: `%4d` sieht mindestens vier Zeichen für den folgenden Ganzzahlwert vor.
5. Die meisten Formatierungen können ein Minuszeichen „`-`“ enthalten, wodurch der folgende Parameter linksbündig ausgegeben wird.

Im Weiteren greifen auf die folgenden Variablendefinitionen und Initialisierungen zurück:

```
1 int    i = 34;
2 double d = 12.34;
3 char   c = 'a';
4 char  *str = "C-Kurs";
5 int   *ip = 0x12345678;
```

Die Ausgabeformatierungen:

Format	Bedeutung	Beispiel	Ausgabe
%c	char	`printf( "%c", c );`	a
	Zeichen	`printf( "%c", '1' );`	1
		`printf( "%c", 49 );`	1 (bei ASCII Kodierung)
		`printf( "%3c", c );`	␣␣a
		`printf( "%-3c", c );`	a␣␣
%s	string	`printf( "%s", str );`	C-Kurs
		`printf( "%s", "bloed" );`	bloed
		`printf( "%8s", str );`	␣␣C-Kurs
		`printf( "%-8s", str );`	C-Kurs␣␣
		`printf( "%3s", str );`	C-Kurs
%d	int	`printf( "%d", i );`	34
%i		`printf( "%i", i );`	34
		`printf( "%4d", i );`	␣␣34
		`printf( "%04d", i );`	0034
		`printf( "%-4d", i );`	34␣␣
		`printf( "%2d", 120+3 );`	123
%x	hexa-	`printf( "%x", i );`	22
%X	dezimal	`printf( "%x", 0x1A1f );`	1a1f
		`printf( "%X", 0x1A1f );`	1A1F
		`printf( "0x%x", 0x1A1f );`	0x1a1f
		`printf( "0x%6x", 0x1A1f );`	0x␣␣1a1f
		`printf( "0x%06x", 0x1A1f );`	0x001a1f
%f	fixed	`printf( "%f", d );`	12.340000
	point	`printf( "%10f", d );`	␣12.340000
		`printf( "%8.3f", d );`	␣␣12.340
%e	expontial	`printf( "%e", d );`	1.234000e+01
		`printf( "%12e", d );`	1.234000e+01
		`printf( "%.3e", d );`	1.234+01
		`printf( "%12.4e", d );`	␣␣1.2340+01
%p	pointer	`printf( "%p", & d );`	0xbff88268
*	width	`printf( "%*d", 4, i );`	␣␣34
		`printf( "%*s", 6, "abc");`	␣␣␣abc
		`printf( "%*.*f", 7, 3, d );`	␣12.340

Anhang D

Kurzfassung der Eingabeformatierungen

An einigen Stellen im Skript wurden auch verschiedene Eingabeformatierungen erwähnt. Auch bei der Eingabe hat man wieder vielfältige Foratierungsmöglichkeiten, von denen hier nur die wichtigsten kurz zusammengefasst werden, da sich der Programmieranfänger auf die einfache Eingabe einzelner Variablen beschränken sollte. Eine vollständige Beschreibung kann man der einschlägigen Dokumentation entnehmen. Unter Linux eignet sich hierfür beispielsweise das Kommando: `man 3 scanf`. In den folgenden Beispielen greifen wir auf die folgenden Variablendefinitionen zurück: `int i; double d; char c;`

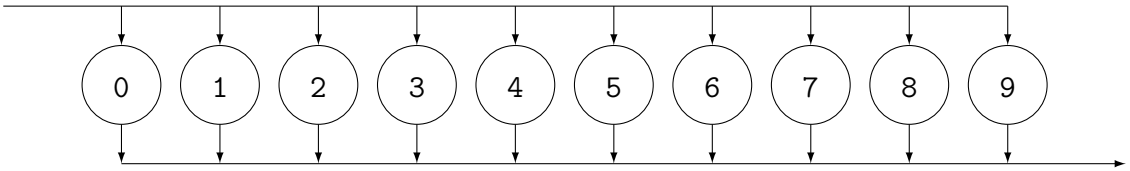
Format	Datentyp	Beispiel
`%c`	`char`	`scanf( "%c", & c );`
`%d`	`int`	`scanf( "%d", & i );`
`%lf`	`double`	`scanf( "%lf", & d );`

Anhang E

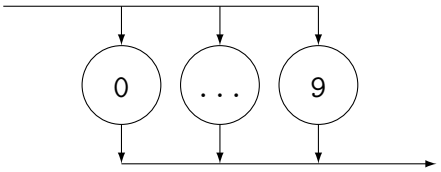
Syntaxdiagramme

Kapitel 20: Syntaxdiagramme: Beispiel Namen (Seite 64)

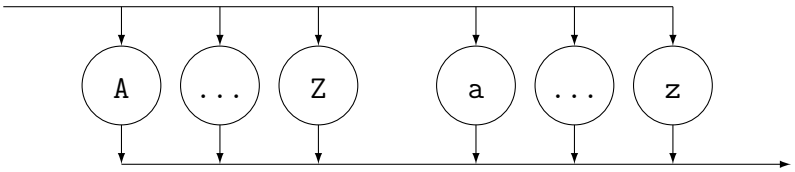
`Ziffer (ausführlich)`



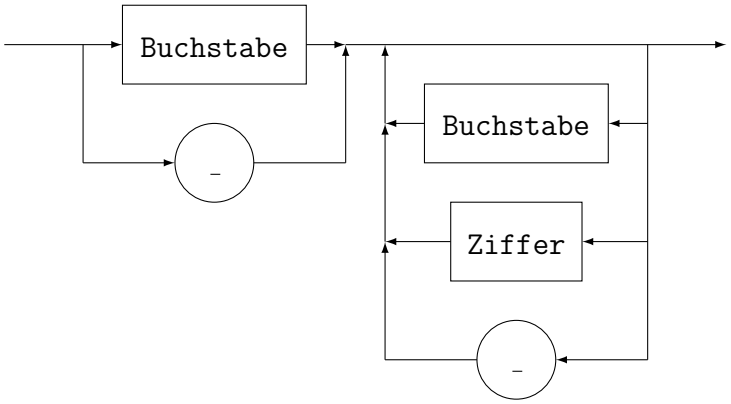
`Ziffer (Kurzform)`



`Buchstabe`

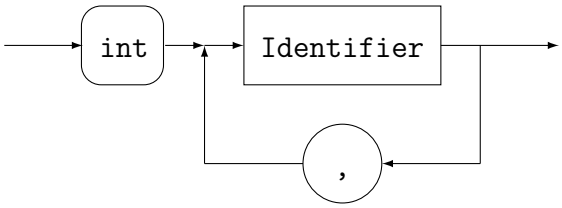


Identifier

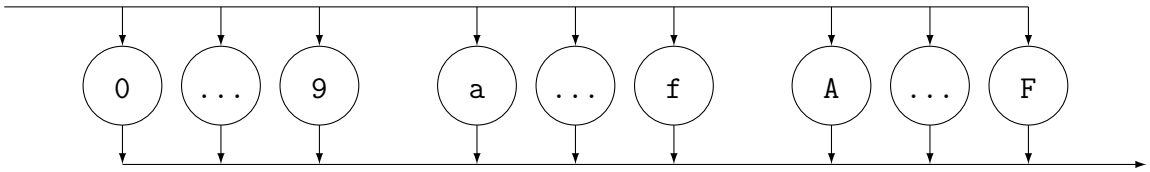


Kapitel 21: Datentyp int für ganze Zahlen (Seite 69)

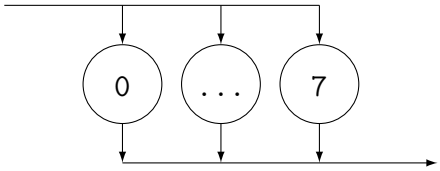
int-Definition (vereinfacht)



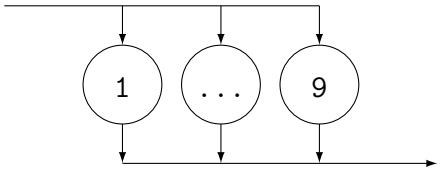
Hex-Ziffer



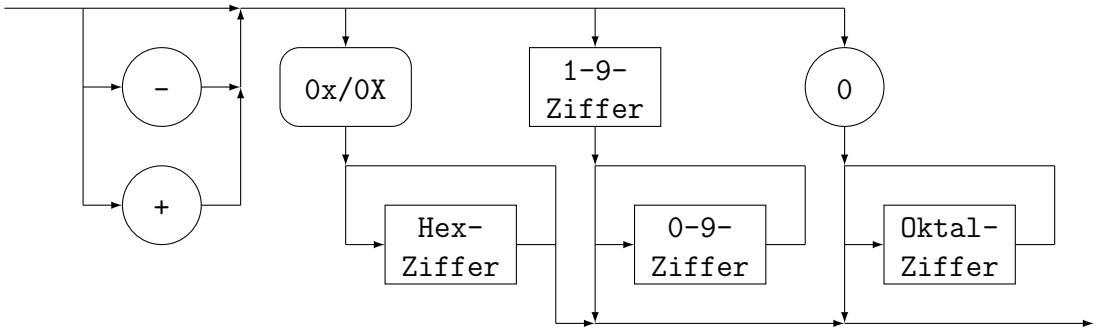
Oktal-Ziffer



1-9-Ziffer

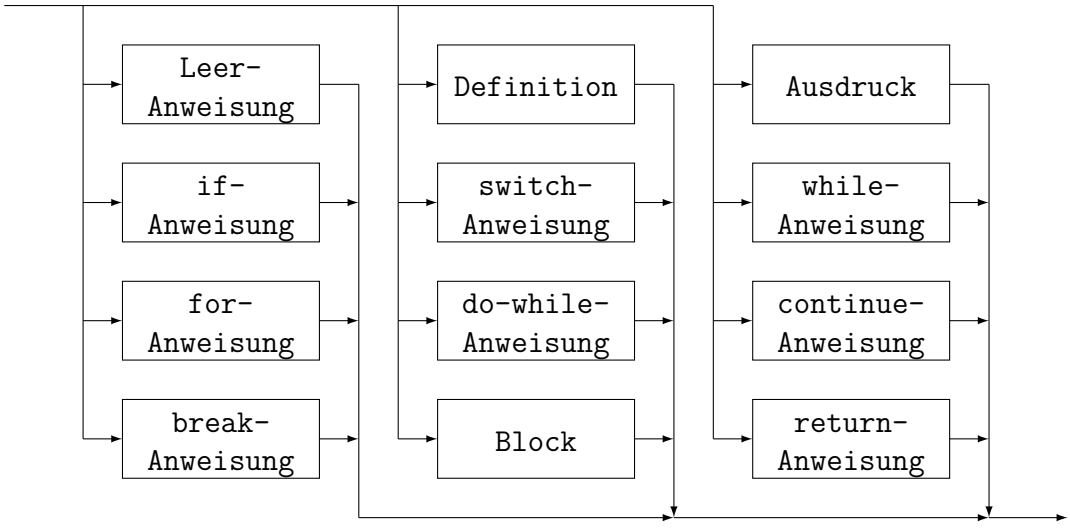


int-Konstante

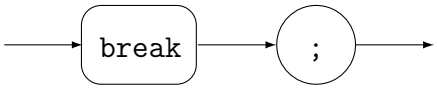


Kapitel 23: Anweisungen, Blöcke und Klammern (Seite 76)

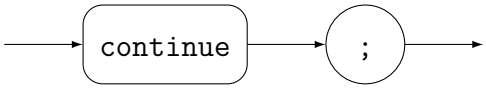
Anweisung



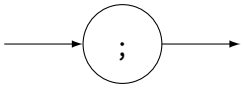
break-Anweisung



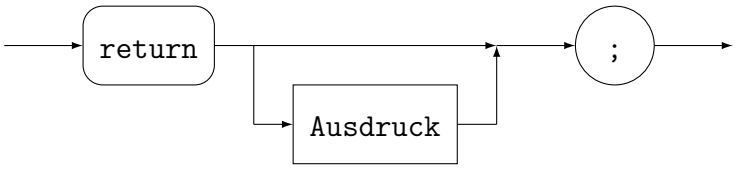
continue-Anweisung



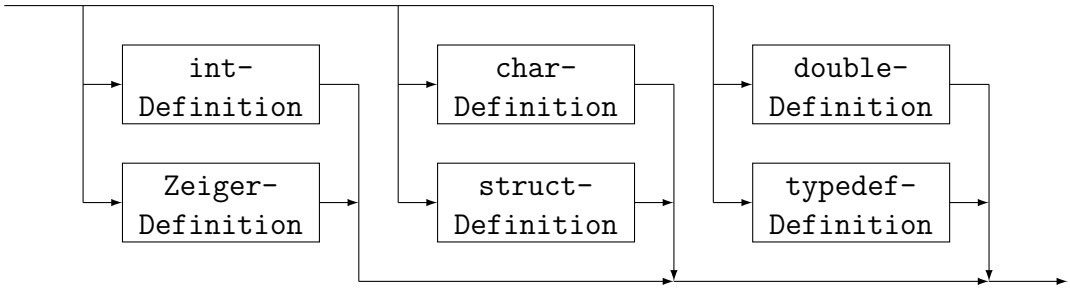
Leer-Anweisung



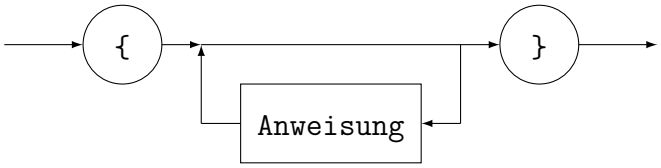
return-Anweisung



Definition

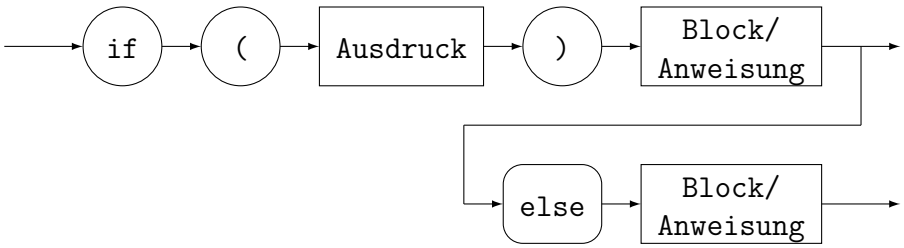


Block



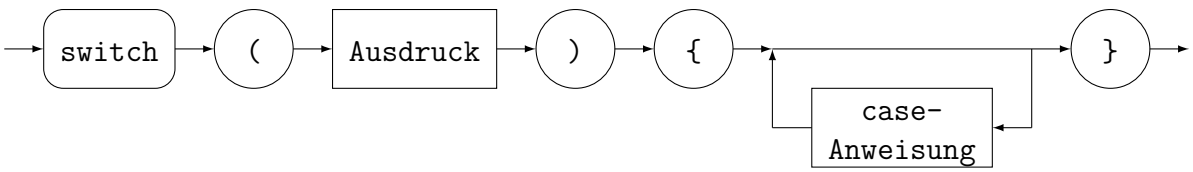
Kapitel 24: Einfache Fallunterscheidung: `if-else` (Seite 80)

if-Anweisung

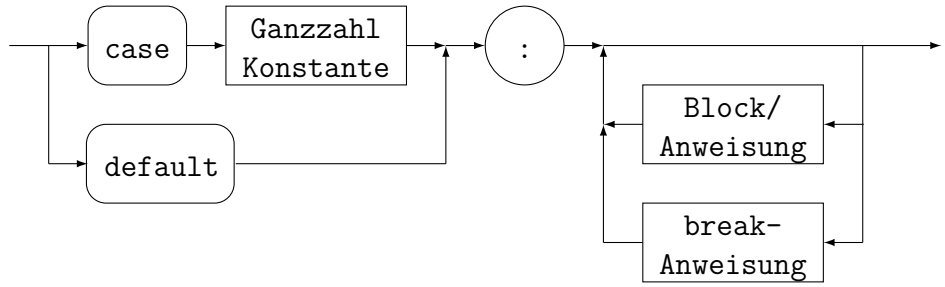


Kapitel 25: Mehrfache Fallunterscheidung: `switch` (Seite 83)

switch-Anweisung

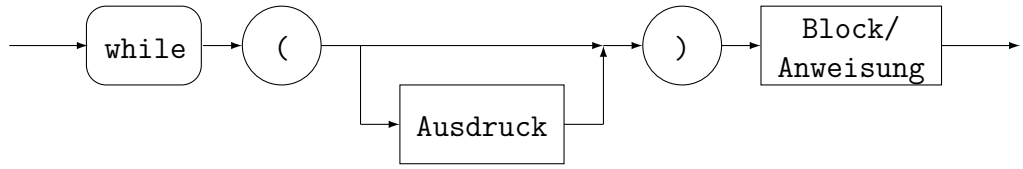


case-Anweisung



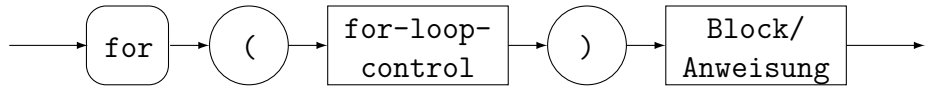
Kapitel 26: Die `while`-Schleife (Seite 88)

while-Schleife

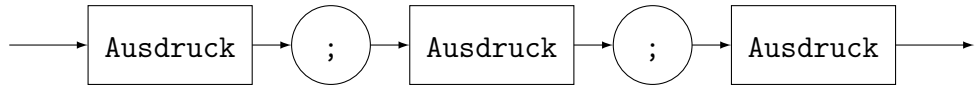


Kapitel 24: Einfache Fallunterscheidung: `if-else` (Seite 91)

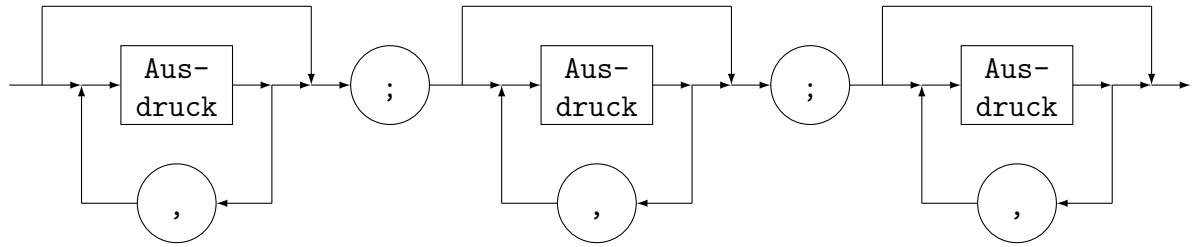
for-Schleife



for-loop-control (vereinfacht)

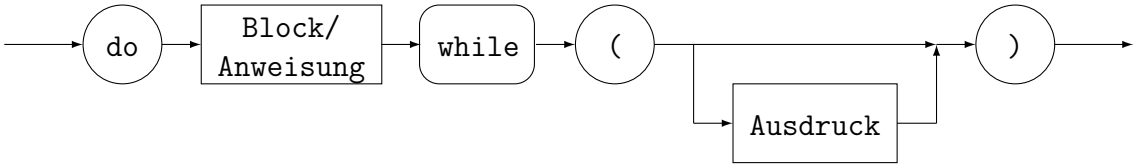


for-loop-control (vollständig)



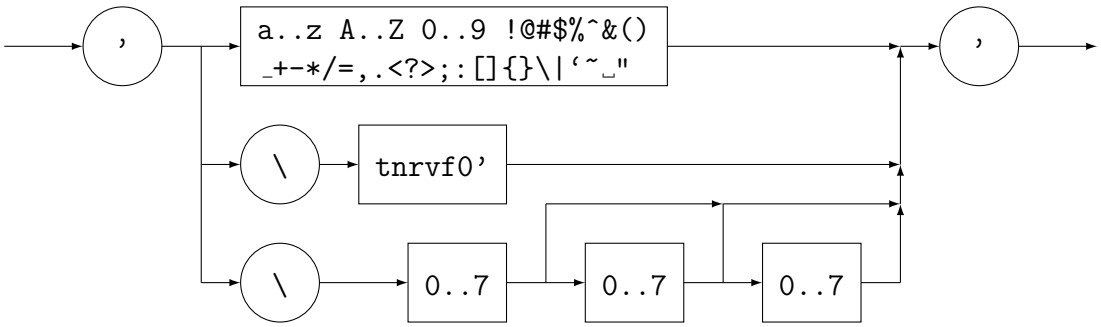
Kapitel 28: Die do-while-Schleife (Seite 95)

do-while-Schleife

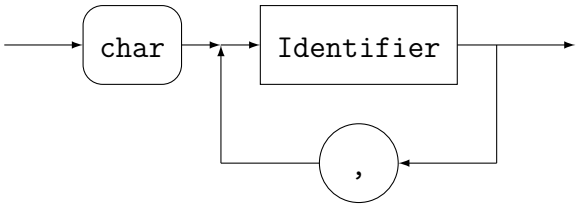


Kapitel 30: Datentyp char: ein einzelnes Zeichen (Seite 101)

char-Konstante

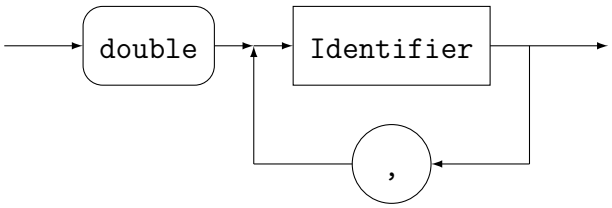


char-Definition (vereinfacht)

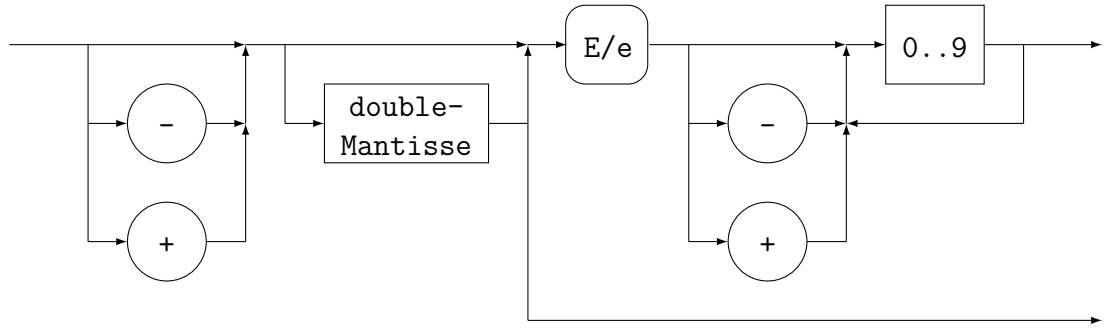


Kapitel 32: Datentyp double für „reelle“ Zahlen (Seite 111)

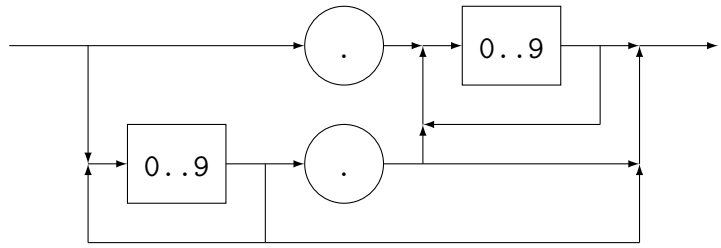
double-Definition (vereinfacht)



double-Konstante

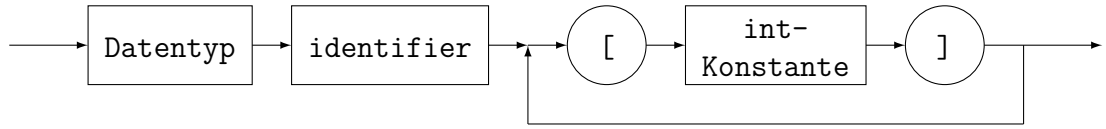


double-Mantisse



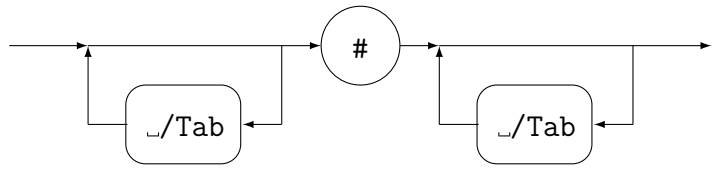
Kapitel 33: Arrays: Eine erste Einführung (Seite 116)

array-Definition

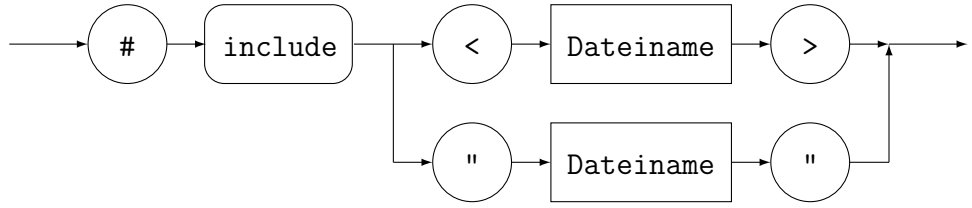


Kapitel 38: Der Präprozessor cpp (Seite 140)

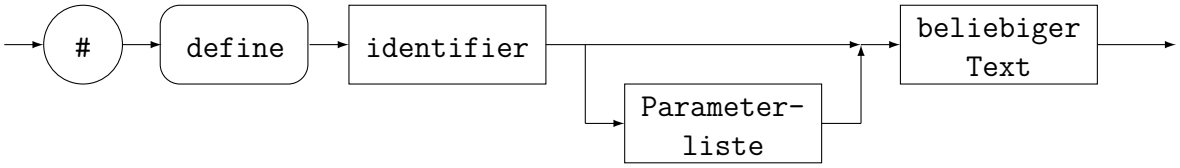
#-Präfix



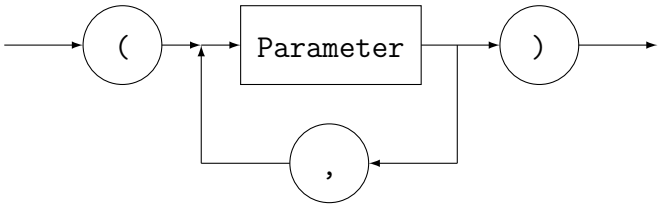
#include-Direktive



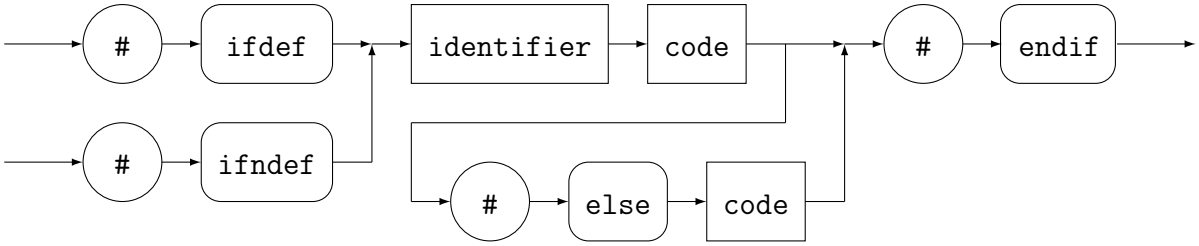
#define-Direktive



#define-Parameterliste

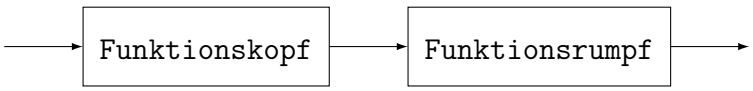


#idef-Direktive

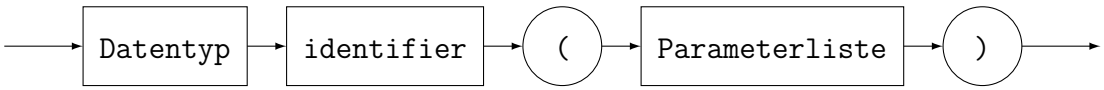


Kapitel 44: Programmierung eigener Funktionen (Seite 186)

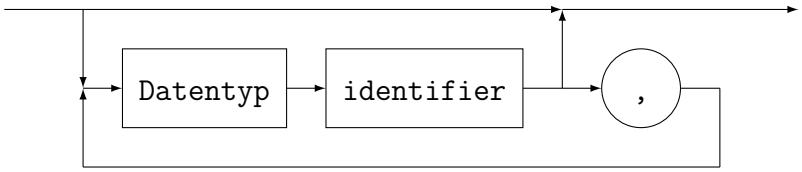
Funktionsdefinition



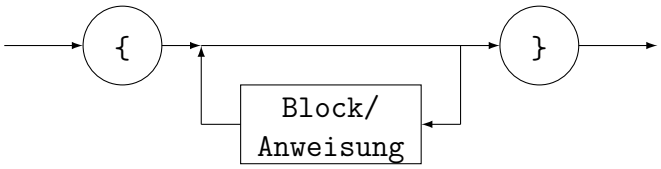
Funktionskopf



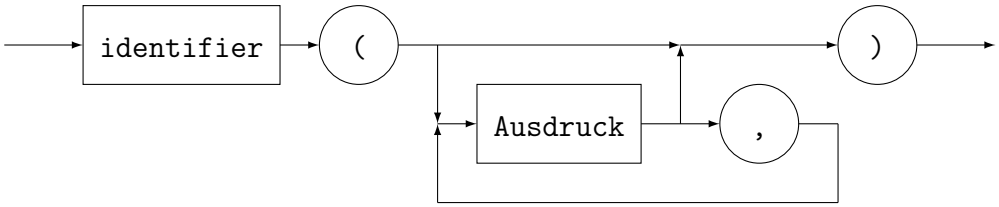
Parameterliste



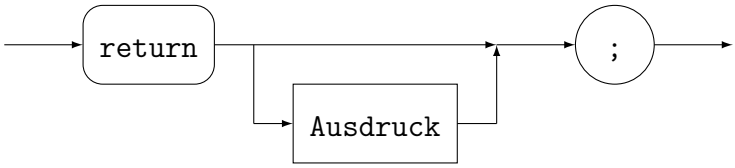
Funktionsrumpf



Funktionsaufruf

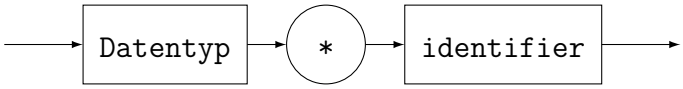


return-Anweisung

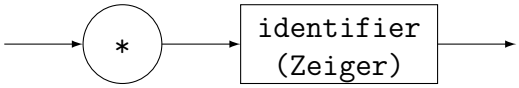


Kapitel 45: Zeiger und Adressen (Seite 200)

Zeiger-Definition

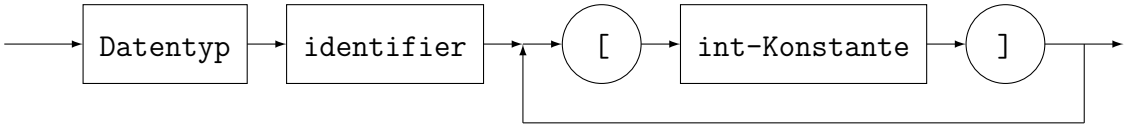


Zeiger Dereferenzieren



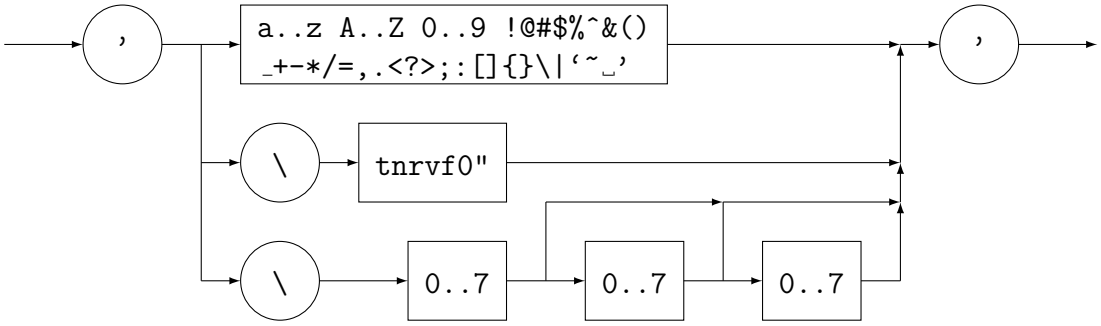
Kapitel 49: Mehrdimensionale Arrays (Seite 223)

Definition n-dimensionales Array

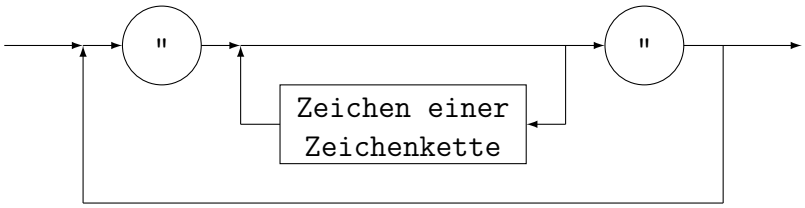


Kapitel 50: Zeichenketten bzw. Datentyp `string` (Seite 228)

Zeichen einer Zeichenkette

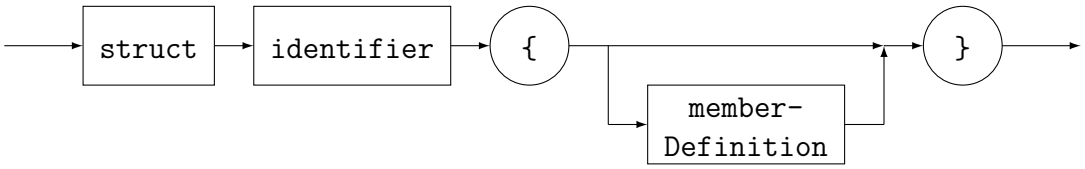


konstante Zeichenkette

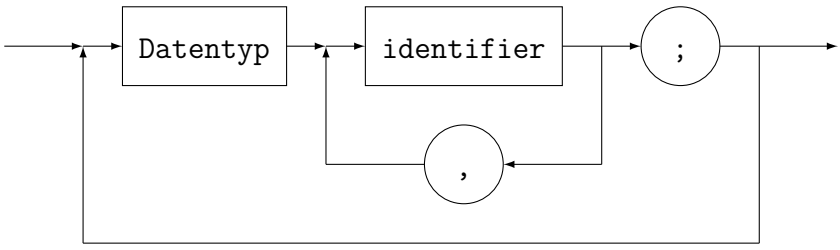


Kapitel 53: Zusammengesetzte Datentypen: `struct` (Seite 253)

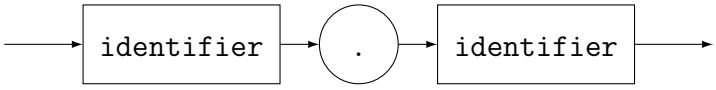
struct



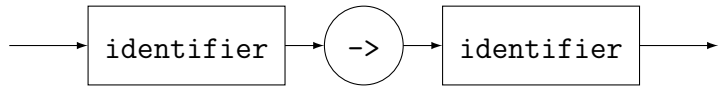
member-Definition



member-Zugriff

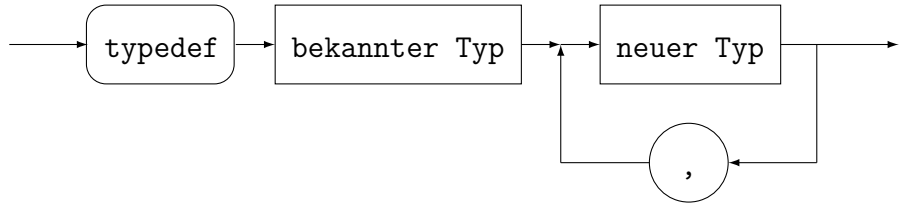


member-Zeiger-Zugriff



Kapitel 54: typedef: Selbstdefinierte Datentypen (Seite 258)

typedef



Literaturverzeichnis

[1] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1990.

[2] Thomas Flik, Hans Liebig und M. Menge. *Mikroprozessortechnik und Rechnerstrukturen.* Springer-Lehrbuch, 1998.

[3] Reinhold Kimm, Wilfried Koch, Werner Simonsmeier, Friedrich Tontsch. *Einführung in Software Engineering.* Walter de Gruyter Verlag, 1979.

[4] Brian W. Kernighan und Dennis Ritchie. *The C Programming Language.* Prentice Hall Software, 1978.

[5] Niklaus Wirth. *Algorithmen und Datenstrukturen.* Vieweg+Teubner Verlag, 1991.

[6] wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange

[7] http://de.wikipedia.org/wiki/Escape-Sequenz.

[8] http://en.wikipedia.org/wiki/C_character_classification.

[9] http://en.wikipedia.org/wiki/Data_segment.

[10] http://de.wikipedia.org/wiki/IEEE_754.

[11] http://de.wikipedia.org/wiki/Unicode.

[12] http://de.wikipedia.org/wiki/Byte-Reihenfolge.

[13] http://en.wikipedia.org/wiki/C99 oder
http://de.wikipedia.org/wiki/Standard_C_Library.

[14] http://www1.icsi.berkeley.edu/~sather/.