Computerorientierte Mathematik

Alexander Martin
Daniel Junglas

Technische Universität Darmstadt

26. Mai 2008

Inhaltsverzeichnis

Kapitel 1

Die Basics

Who's General Failure and
why's he reading my disk?
— Anon

In diesem ersten Kapitel wollen wir einige Grundlagen erläutern, die für das Arbeiten mit dem Rechner während dieser Veranstaltung wichtig sind. Wir beschreiben dazu zuerst die Hardware so detailliert, wie wir sie betrachten wollen. In den weiteren Abschnitten geben wir dann eine kurze Einführung in LINUX-System im Allgemeinen und in das System am Rechnerpool des Fachbereichs Mathematik in Darmstadt im Einzelnen.

1.1 Hardware

So wie wir den Computer in dieser Vorlesung betrachten, besteht er im Wesentlichen aus fünf funktionellen Einheiten, die zum Daten- und Kommandoaustausch über einen oder mehrere **Busse** (*engl.* ,,buses") miteinander verbunden sind. Diese Einheiten sind (siehe auch Abbildung 1.1)

1. die **CPU** (**C**entral **P**rocessing **U**nit),
2. der **Hauptspeicher** (*engl.* ,,(main) memory"),
3. der **Bildschirm** (*engl.* ,,monitor"),
4. die **Tastatur** (*engl.* ,,keyboard") sowie
5. die **Festplatte** (*engl.* ,,hard disk").

Über die Busse (oder zusätzliche weitere Busse) kann auch noch weitere Hardware wie z.B. CD-Laufwerke an den Computer angeschlossen werden. Wir wollen nun die einzelnen Bestandteile etwas genauer erläutern:

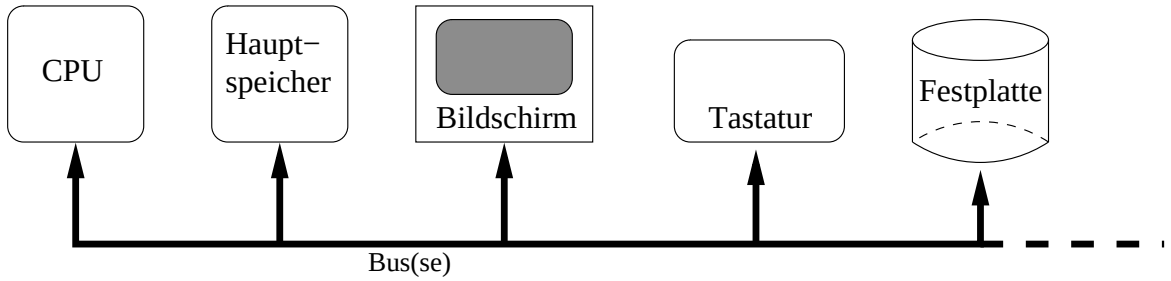


Abbildung 1.1: Hardware im Computer.

1.1.1 CPU

Die CPU (**C**entral **P**rocessing **U**nit) ist das ,,Arbeitstier" im Computer. Die CPU führt Anweisungen aus, die ihr in Maschinensprache (das ist eine Folge von Nullen und Einsen) aufgetragen werden. Um dies möglichst effizient zu tun, verfügt die CPU über einen kleinen Speicher, bestehend aus den sogenannten **Registern** (*engl.* ,,registers"), auf die sehr schnell zugegriffen werden kann. Diese Register sind für uns im Moment nicht von Bedeutung, wir werden aber zu gegebener Zeit darauf zurückkommen.

1.1.2 Hauptspeicher

Der Hauptspeicher ist eine Ansammlung von mehreren Millionen sogenannten **Bits**. Ein Bit kann den Wert 0 oder 1 haben, was durch ,,Strom an" bzw. ,,Strom aus" realisiert wird. Die Bits sind alle hintereinander geordnet, sodass der Hauptspeicher als eine lange Kette von Bits angesehen werden kann.

Typischerweise sind jeweils acht aufeinanderfolgende Bits zu einem **Byte** zusammengefaßt. Da ein Byte also aus acht Bits besteht, kann ein Byte genau 2^8 verschiedene Zustände annehmen. So lassen sich mit einem Byte die Zahlen von 0 bis 255 darstellen, indem man ein Byte als achtstellige binäre Zahl (möglicherweise mit führenden Nullen) betrachtet. Möchte man grössere ganze Zahlen darstellen, so muss man noch einmal mehrere Bytes zusammenfassen.

Der Hauptspeicher dient zwei Zwecken:

- Programme können in den Hauptspeicher geladen werden. Da der Hauptspeicher deutlich schneller ausgelesen werden kann als z.B. eine Festplatte (siehe Abschnitt 1.1.3 unten), wird ein Programm vor der Ausführung meist komplett in den Hauptspeicher eingelesen. Liegt es erst einmal im Hauptspeicher, kann es deutlich schneller als von der Festplatte ausgeführt werden. Das ist insbesondere dann von Vorteil, wenn das Programm oder Teile von ihm mehrfach hintereinander ausgeführt werden sollen.

- Programme können Daten im Hauptspeicher ,,abspeichern". Ein Programm kann einen Teil des Hauptspeichers ,,reservieren" und auf den reservierten Bytes beliebige Daten ablegen. Diese Daten bleiben dort gespeichert, bis sie vom Programm mit

anderen Daten überschrieben werden, das Programm die reservierten Bytes wieder freigibt (dies geschieht spätestens dann, wenn das Programm beendet wird) oder der Computer ausgeschaltet wird.

Hauptspeicher in Personal-Computern haben zur Zeit eine Grösse, die von mehreren Hundert Mega-Byte[1] bis zu wenigen Giga-Byte reicht.

1.1.3 Festplatte(n)

Die Festplatte dient wie der Hauptspeicher zum Speichern von Daten und Programmen. Im Gegensatz zum Hauptspeicher gehen hier die Daten aber nicht beim Beenden des Programms oder beim Ausschalten des Computers verloren, die Daten und Programme können also persistent gespeichert werden und werden erst gelöscht, wenn sie explizit entfernt oder überschrieben werden. Dieser Vorteil gegenüber dem Hauptspeicher bringt aber auch einen gravierenden Nachteil mit sich: Festplatten sind deutlich langsamer als Hauptspeicher.

Um die Arbeit mit Festplatten möglichst einfach und effizient zu gestalten, sind auf Festplatten meistens sogenannte **Dateisysteme** (*engl.* ,,filesystem") (siehe dazu Abschnitt 1.3 unten) angelegt. Mit Hilfe dieser Dateisysteme können die gespeicherten Daten und Programme als **Dateien** (*engl.* ,,files") repräsentiert und in **Verzeichnissen** (*engl.* ,,directories") organisiert werden.

Festplatten in Personal-Computern haben heutzutage meist eine Kapazität von mehreren zig bis hundert Giga-Byte.

1.1.4 Busse

Die einzelnen Hardware-Komponenten des Computers sind durch sogenannte ,,Busse" miteinander verbunden. Diese Busse dienen zum Austausch von Daten und Kommandos zwischen den verschiedenen Komponenten. Möchte z.B. die CPU den Wert eines bestimmten Bytes aus dem Hauptspeicher erfahren, so schreibt sie den Befehl ,,lies Byte xyz" auf den entsprechenden Bus. Der Hauptspeicher (oder genauer die **MMU**, die **M**emory **M**anagement **U**nit) liest den Befehl vom Bus, sucht das ensprechende Byte und schreibt seinen Wert auf den Bus. Vom Bus kann die CPU dann wiederum den Wert lesen.

Anders als in Abbildung 1.1 dargestellt, gibt es im Computer nicht nur einen, sondern eine ganze Reihe von Bussen, die parallel bzw. gleichzeitig benutzt werden können. Auf diese Art läßt sich die Effizienz des Systems deutlich erhöhen. Beispiele für solche Busse sind der ,,Universal Serial Bus" (USB), der ,,Peripheral Component Interconnect (PCI) Bus" oder der ,,SCSI Bus".

1.1.5 Bildschirm/Tastatur

Bildschirm und Tastatur sind die klassischen Aus- und Eingabegeräte am Computer. Mit Hilfe der Tastatur teilt man dem Computer mit, was man von ihm will und mit Hilfe des

[1] 1 KB (Kilo-Byte) = 2^{10} Byte, 1 MB (Mega-Byte) = 2^{10} Kilo-Byte, 1 GB (Giga-Byte) = 2^{10} Mega-Byte.

Bildschirms lassen sich Ergebnisse und Rückmeldungen visualisieren.

Heutzutage verfügen die meisten Computer auch noch über ein sogenanntes ,,Zeigegerät" (Maus, Trackball, ...), das die Eingabe von Daten und Kommandos noch weiter erleichtert, insbesondere wenn man eine graphische Benutzeroberfläche zur Verfügung hat.

1.2 Betriebssystem

Wie bereits in Abschnitt 1.1.1 erwähnt versteht die CPU nur Maschinensprache. Hinzu kommt, das z.B. je nach Typ und Hersteller der Festplatte *verschiedene* Kommandos bzw. Kommandoformate (in Maschinensprache) notwendig sind, um Daten mit der Festplatte auszutauschen. Diese Vielfalt macht die Arbeit mit dem Computer mühselig und ineffizient.

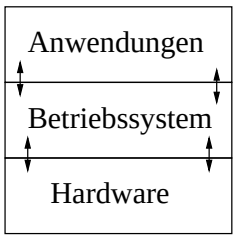


Abbildung 1.2: Zwiebelschalenmodell für das Betriebssystem.

Das Betriebssystem dient dazu, diese Probleme zu beheben. Zu diesem Zweck stellt es für die häufig durchgeführten Aktionen (wie z.B. Daten auf die Festplatte zu schreiben) sogenannte Funktionen (auch ,,Syscalls" genannt) zur Verfügung. Ein Anwendungsprogramm kann dann z.B. die Funktion ,,Datei erstellen" aufrufen, ohne Typ und Hersteller der Festplatte zu kennen. Das Betriebssystem trägt in diesem Fall Sorge (und Verantwortung) dafür, dass auf der Festplatte tatsächlich eine entsprechende Datei erstellt wird. Hierbei ist zu beachten, dass auch die Syscalls in Maschinensprache implementiert sind und deswegen auch in dieser aufzurufen sind (dies macht eine Shell erforderlich – siehe unten).

Durch die Benutzung des Betriebssystems lassen sich Anwendungen also so programmieren, dass sie nur das Betriebssystem und nicht Hardware eines bestimmten Typs oder Herstellers voraussetzen.

Eine weitere Aufgabe des Betriebssystems ist die korrekte Initialisierung der Komponenten beim Start des Computers.

Ausserdem übernimmt das Betriebssystem noch eine Reihe von Verwaltungsaufgaben. Zu diesen Aufgaben gehört z.B. das Erfassen von Laufzeit, Grösse und Speicherverbrauch aller laufenden Anwendungen.

Einige der bekannteren Dateisysteme sind z.B. WINDOWS©, UNIX, LINUX, SUNOS oder MACOS. Da am Rechnerpool der TU Darmstadt LINUX installiert ist, werden wir uns im Weiteren ausschließlich mit diesem Betriebssystem befassen.

Wie bereits erwähnt, ist das Betriebssystem für eine sinnvolle Organisation der im Computer gespeicherten Daten verantwortlich. Im Normalfall organisiert das Betriebssystem

diese Daten in Dateien und Ordnern, die zusammengenommen ein sogenanntes Dateisystem ergeben. Mit diesem Dateisystem wollen wir uns nun etwas näher beschäftigen.

1.3 Dateisystem

Unter LINUX wird das Dateisystem durch einen sogenannten ,,Baum" dargestellt, der allerdings auf dem Kopf steht (siehe Abbildung 1.3):

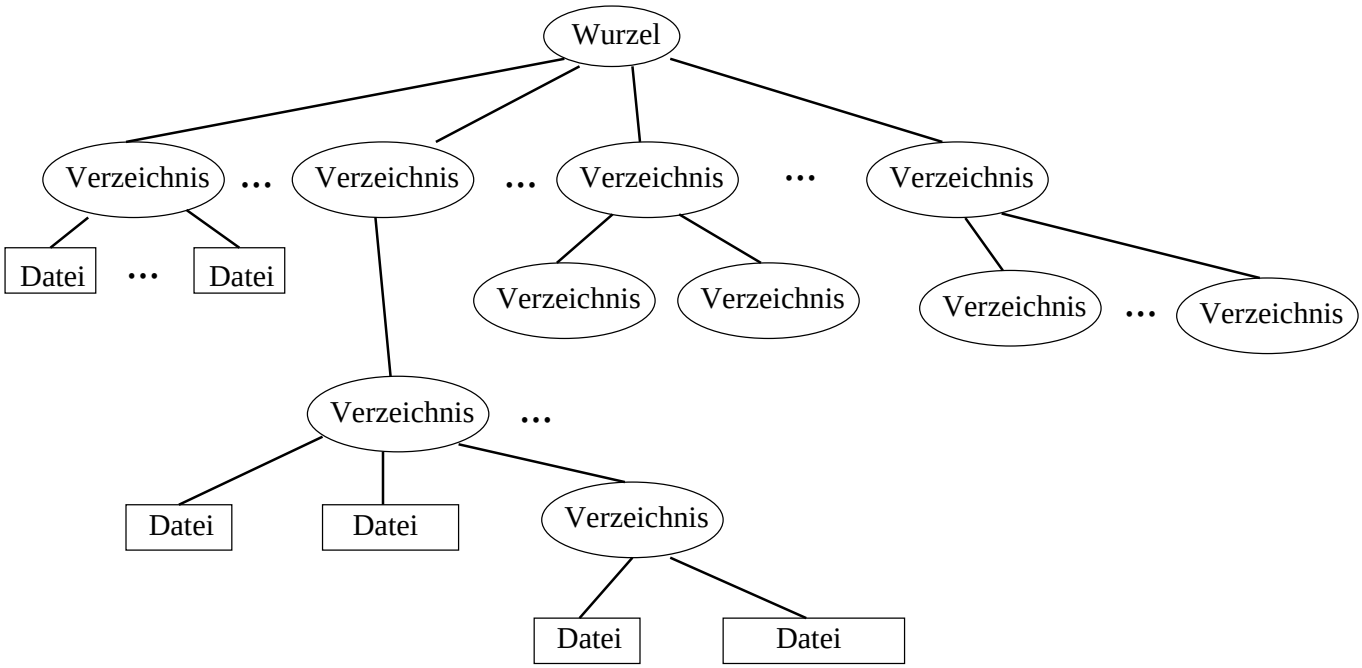


Abbildung 1.3: Schematische Darstellung des Verzeichnisbaums.

- Es gibt ein **Wurzelverzeichnis** (*engl.* ,,Root Directory"), das über allen anderen Verzeichnissen steht.
- Von dem Wurzelverzeichnis gehen mehrere Verzweigungen ab, von denen jede ein Verzeichnis (*engl.* ,,Directory") darstellt.
- Ein Verzeichnis kann entweder leer sein oder selbst weitere Verzeichnisse (d.h. Verzweigungen) und/oder Dateien (das sind dann die Blätter) enthalten.

Dateien und Verzeichnisse werden mit Namen bezeichnet, in denen alle Zeichen ausser dem **Schrägstrich** (*engl.* ,,Slash") erlaubt sind. Die einzige Ausnahme bildet hierbei das Wurzelverzeichnis, das den Namen ,,`/`" trägt. Die erlaubte Länge eines Dateinamens beträgt normalerweise 4096 Zeichen.

Abbildung 1.4 zeigt einen Ausschnitt aus einem möglichen Dateibaum: Direkt unter der Wurzel finden sich die Verzeichnisse `bin`, `home`, `www` und `usr`, wobei das erste die Dateien `ls`

und `date` enthält. Das Verzeichnis `home` enthält ein weiteres Verzeichnis, nämlich `junglas`. In diesem Verzeichnis befinden sich zwei Dateien (nämlich `.bashrc` und `CoMa.txt`) sowie ein weiteres Unterverzeichnis (`data`).

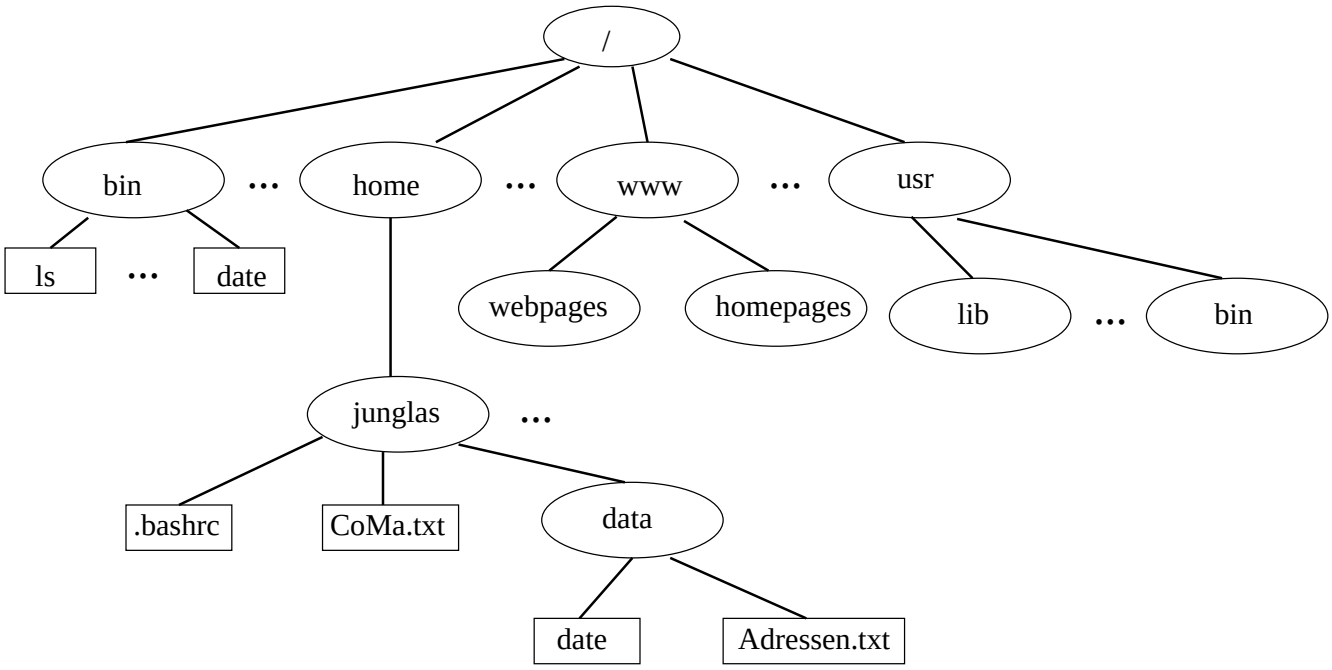


Abbildung 1.4: Ausschnitt aus einem (möglichen) LINUX-Dateibaum.

Da das Dateisystem durch einen Baum dargestellt werden kann, gibt es für jede Datei und jedes Verzeichnis einen *eindeutigen* (kürzesten) **Weg** (*engl.* „Path") vom Wurzelverzeichnis zu dieser Datei/diesem Verzeichnis. Verbindet man alle Namen, die auf diesem Pfad liegen mit einem Schrägstrich, so erhält man einen eindeutigen Namen, den **Pfadnamen** (*engl.* „Pathname"). Der Pfadname für die Datei `ls` aus Abbildung 1.4 ist demnach `//bin/ls`. Der Einfachheit halber, werden mehrere aufeinanderfolgende Schrägstriche in solchen Namen zu einem einzigen Schrägstrich zusammengefaßt, d.h. die Namen `//bin/ls` und `/bin/ls` sind äquivalent. Mit dieser Erläuterung ist klar, dass das Beispiel in Abbildung 1.4 die Verzeichnisse

```
/
/bin
/home
/home/junglas
/home/junglas/data
/www
/www/webpages
/www/homepages
/usr
/usr/lib
/usr/bin
```

sowie die Dateien

```
/bin/ls
/bin/date
/junglas/.bashrc
/junglas/CoMa.txt
/junglas/data/date
/junglas/data/Adressen.txt
```

enthält. Insbesondere können also die beiden gleichnamigen `date`-Dateien in den Verzeichnissen `/bin` und `/home/junglas/data` anhand ihres Pfadnamens unterschieden werden.

Übung: .....

1. spezielle Dateisysteme: procfs, devfs,
2. INodes,
3. Hardlinks,
4. Symlinks

.....

1.4 Shell

Obwohl das Betriebssystem bereits den Zugriff auf die Hardware und die Benutzung des Computers deutlich erleichtert, ist es meist noch recht schwierig, das Betriebssystem direkt zu benutzen. Auch dies muss nämlich im Normalfall in Maschinensprache geschehen. Damit der Mensch nicht in Maschinensprache mit dem System kommunizieren muss, gibt es die Shell. Wie Abbildung 1.5

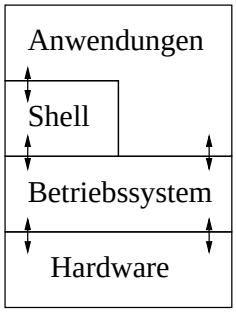


Abbildung 1.5: Zwiebelschalenmodell für die Shell.

zeigt, ist die Shell eine weitere Abstraktionsebene über dem Betriebssystem.

Tatsächlich ist die Shell ein interaktives Programm, das jeweils auf Tastatur-Eingaben vom Anwender wartet. Über die Tastatur kann der Benutzer der Shell Kommandos erteilen, die dem Schema in Abbildung 1.6 folgen:

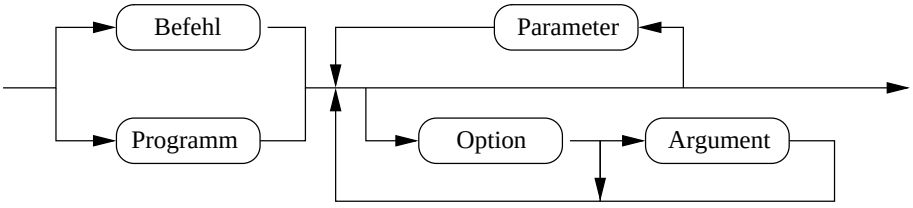


Abbildung 1.6: Schematischer Aufbau eines Shell-Kommandos.

1. Jedes Kommando beginnt entweder mit einem Shell-internen Befehl oder dem Namen des Programms, das ausgeführt werden soll.

2. Dem Befehl/Programmname folgen keine, eine oder mehrere **Optionen** (*engl.* ,,Options"), die zum genaueren Steuern der Befehle/Programme verwendet werden. Solche Optionen beginnen unter LINUX meist mit einem oder zwei Minuszeichen (,,-").

3. Eine Option kann zusätzlich noch ein **Argument** (*engl.* ,,argument") haben.

4. Nach den Optionen folgen dann eine (möglicherweise leere) Menge von **Parametern** (*engl.* ,,parameters"), die ebenfalls zur Steuerung des Befehls/Programms dienen.

Hinweis für die Vorlesung:
Optionen, Parameter, Argumente sind nicht eindeutig belegt und werden häufig vermischt.

(In der Realität sind auch leichte Abweichungen von diesem Muster möglich, wir werden uns in der Vorlesung aber bemühen, uns daran zu halten.)

Einzelne Teile der Kommandos müssen dabei durch eine beliebige nichtleere Menge von Leerzeichen oder Tabulatoren getrennt werden. Abgeschlossen wird das Kommando mit der **Eingabe**-Taste (*engl.* ,,Enter").

Ein Beispiel für ein solches Kommando ist

```
rm -i Datei
```

Dieses Kommando ruft das Programm `rm` mit der Option `-i` und dem Parameter `Datei` auf (in diesem Fall ist `Datei` kein Argument für `-i`!). Das angegebene Kommando löscht (`rm` für englisch **rem**ove) die Datei `Datei` und zwar interaktiv (deswegen `-i`), d.h. bevor `Datei` wirklich gelöscht wird, wird der Benutzer gefragt, ob er das wirklich tun will.

Um anzuzeigen, dass der vorherige Befehl ausgeführt wurde und die Shell nun bereit für den nächsten ist, zeigt sie die **Eingabeaufforderung** (*engl.* ,,Prompt") an. Diese sieht z.B. so aus

```
fb04305:~ >
```

Was die einzelnen Bestandteile der Eingabeaufforderung bedeuten, werden wir in der Übung sehen.

Betrachten wir uns kurz eine Beispielsitzung in der Shell:

```
fb04305:~ > cp Datei1 Datei2
fb04305:~ > rm Datei1
fb04305:~ >
```

In diesem Beispiel ist die Eingabeaufforderung ,,`fb04305:~ >`". Als erstes Kommando wurde die Datei `Datei1` nach `Datei2` kopiert (`cp` steht für englisch ,,copy"). Nachdem die Shell das Kopierprogramm aufgerufen und erfolgreich beendet hat, zeigt sie mit der erneuten Ausgabe der Eingabeaufforderung an, dass sie bereit für den nächsten Befehl ist. Als nächstes Kommando wurde dann `rm Datei1` eingegeben, d.h. `Datei1` sollte gelöscht werden (`rm` steht für englisch ,,remove"). Nachdem auch dies erfolgreich geschehen ist, ist die Shell mit der Anzeige der Eingabeaufforderung wieder für die nächste Aktion bereit.

An diesem Beispiel ist bereits das ,,no news is good news" der meisten LINUX- bzw UNIX-Programme zu erkennen: eine erfolgreich beendete Aktion wie z.B. das Kopieren einer Datei, wird nicht mit einer Meldung wie ,,Datei1 erfolgreich nach Datei2 kopiert" abgeschlossen. Stattdessen wird eben einfach gar nichts ausgegeben, um zu zeigen, dass es keine Problem gab. Tritt dagegen ein Fehler auf, so wird dies entsprechend angezeigt. Geben wir in obigen Beispiel z.B. noch einmal `rm Datei1` ein, so erhalten wir

```
fb04305:~ > rm Datei1
rm: Entfernen von "Datei1" nicht möglich: Datei oder Verzeichnis nicht
gefunden
fb04305:~ >
```

Da die Datei bereits gelöscht war, kann sie nicht noch ein zweites Mal gelöscht werden.

Nachdem nun klar ist, was eine Shell ist, erläutern wir im Folgenden, wie man diese Shell zum Arbeiten am LINUX-System des Rechnerpools benutzt. Dabei beschränken wir uns im Folgenden auf die `bash` – die ,,bourne again shell". Neben dieser Shell gibt es unter LINUX noch andere häufige Shells, wie z.B.

- **msh**: Minix-Shell,
- **ash**: ,,a shell",
- **lash**: ,,lame ass shell",
- **csh/tcsh**: ,,C shell",
- **zsh**: ,,Z shell",
- **ksh**: ,,korn shell".

Alle diese Shells funktionieren in weiten Zügen analog zur `bash`, sind aber meist nicht so komfortabel wie diese (was z.B. Namen wie ,,lame ass shell" motiviert).

Übung: .....

1. Wie bastelt man sich ein Prompt?
2. Was bedeuten die einzelnen Sachen im Prompt?

.....

1.5 Arbeiten am Linux-System des Rechnerpools

Nachdem wir in den vorigen Abschnitten grob die Struktur des Computers und die Aufgaben eines Betriebssystems erläutert haben, wollen wir nun zeigen, wie man mit dem LINUX-System am Mathematik-Rechnerpool arbeitet.

1.5.1 Anmeldung am System

LINUX ist ein sogenanntes ,,Mehrbenutzer-Betriebssystem". Das bedeutet, dass es zwischen verschiedenen Benutzern am gleichen Rechner/System unterscheidet. Jeder für das System zugelassene Benutzer erhält einen Benutzernamen, eine Benutzernummer (die ,,User ID") und ein Heimatverzeichnis (das ,,Home-Verzeichnis"). Ausserdem erhält jeder Benutzer für den Zugang zum System ein Passwort. So ist sichergestellt, dass sich kein Benutzer unter dem Namen eines anderen Benutzers anmelden kann und in dessen Namen (verbotene) Dinge tun kann.

Die Computer im Rechnerpool am Fachbereich Mathematik warten im Normalfall mit einer Eingabemaske auf, wie sie in Abbildung 1.7 dargestellt ist. Um sich beim System anzumelden,

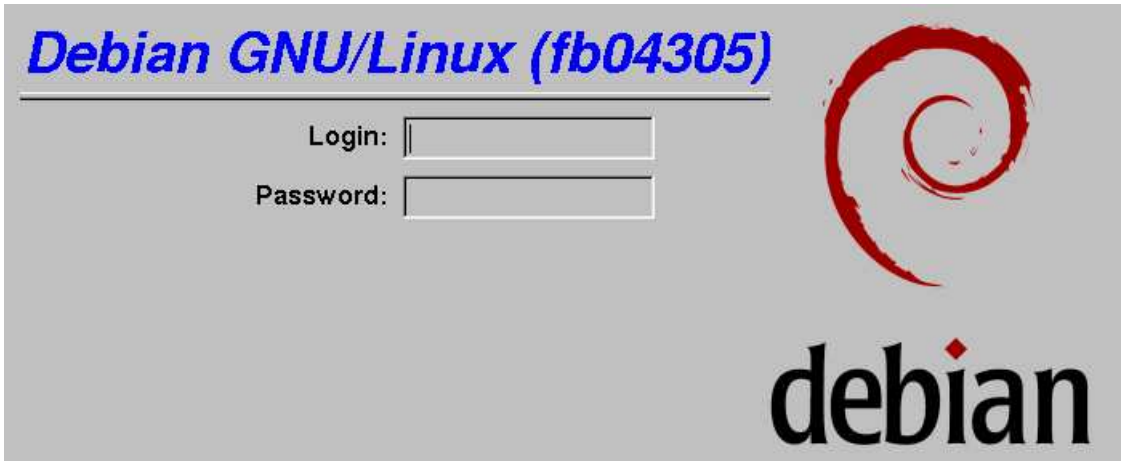


Abbildung 1.7: Login-Bildschirm am Rechnerpool Mathematik.

muss man seinen Benutzernamen sowie sein Passwort eingeben und beide Eingaben mit ENTER abschliessen. Aus Sicherheitsgründen werden bei der Eingabe des Passworts keine Zeichen angezeigt. Man muss sich also selber merken, welchen Teil des Passworts man schon eingegeben hat und welchen man noch eingeben muss. Dafür kann aber eben auch niemand das Passwort vom Bildschirm ablesen.

1.5.2 Der Fenstermanager

Hat man sich erfolgreich am System angemeldet, so erscheint als nächstes der Fenstermanager. Dieser stellt eine graphische Benutzeroberfläche bereit, die das Arbeiten deutlich erleichtert. Abbildung 1.8 zeigt, wie sich der Fenstermanager normalerweise nach dem Einloggen präsentiert. Wir beschreiben zunächst einmal die drei Hauptbestandteile, wie sie in Abbildung 1.8 dargestellt sind:

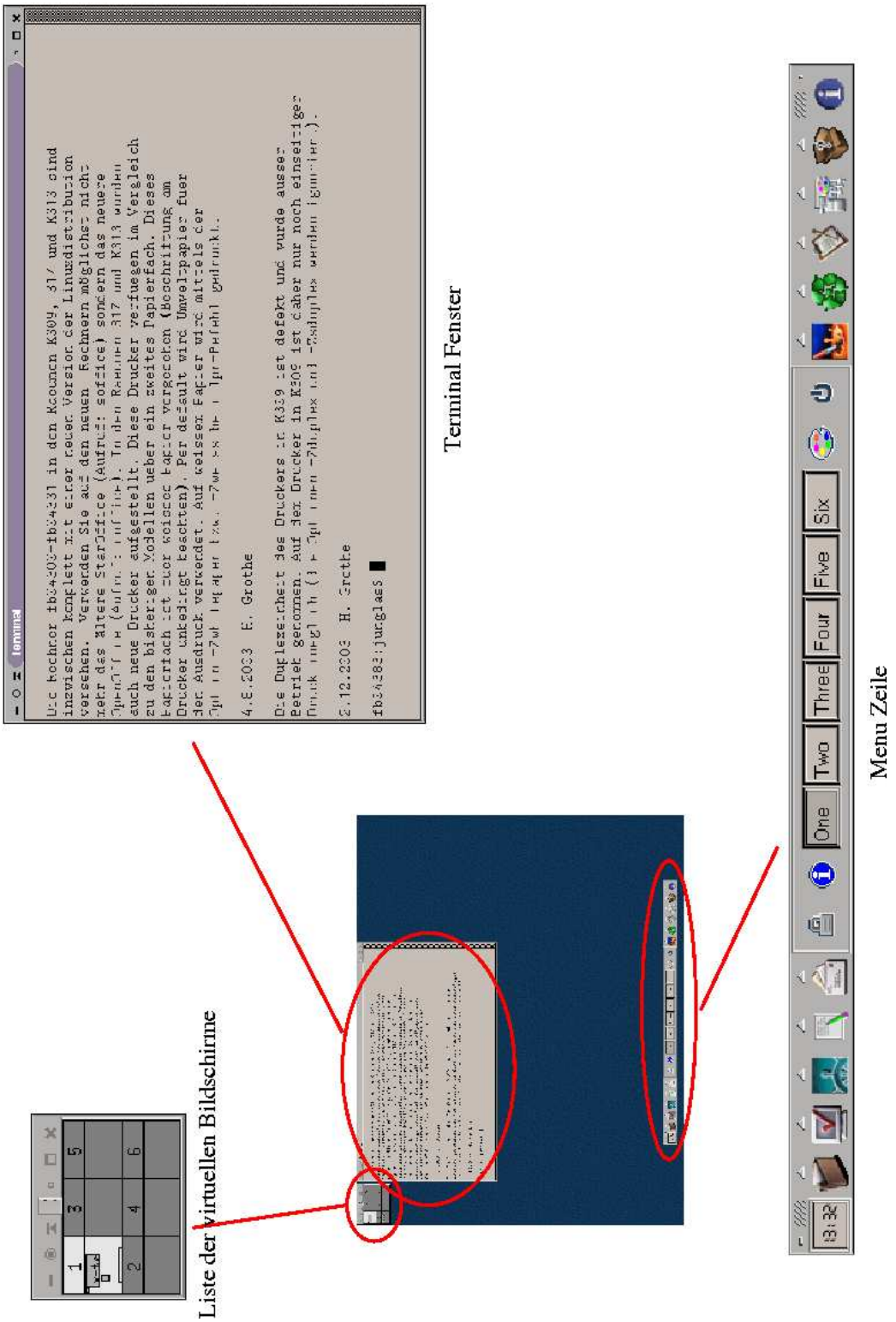


Abbildung 1.8: Der Fenstermanager direkt nach dem Login.

Menüzeile In dieser Zeile sind eine Reihe von Knöpfen angeordnet, mit denen sich einige Operationen beschleunigt ausführen lassen: anstatt das Kommando über die Tastatur einzugeben, klickt man einfach auf den entsprechenden Knopf und das Programm wird gestartet.

Terminalfenster Dieses (erste) Fenster wird normalerweise nach dem Einloggen automatisch geöffnet. Sollte dies nicht der Fall sein, so kann man durch Klicken des in Abbildung 1.9 dargestellten Knopfes ein solches Fenster ,,per Hand" öffnen. Über den selben Knopf kann

Abbildung 1.9: Der Knopf zum Öffnen eines Terminalfensters in der Menüzeile.

man auch weitere Terminalfenster *zusätzlich* öffnen.

In jedem Terminalfenster, das man auf diese Weise öffnet, wird direkt eine Shell gestartet, mit der man dann arbeiten kann. Hat man mehrere Terminals gleichzeitig geöffnet, dann beziehen sich Tastatureingaben immer auf das *aktuelle* Terminal. Dies ist zunächst einmal das zuletzt geöffnete Fenster. Allerdings gibt es zwei Möglichkeiten, ein anderes Terminalfenster zum aktuellen Fenster zu machen:

1. Klickt man in ein Terminal (oder genauer ein Fenster), so erhält dieses den ,,Fokus", d.h. es wird in den Vordergrund geholt (und überlappt dadurch möglicherweise andere Fenster) und folgende Tastatureingaben beziehen sich auf dieses Fenster.
2. Es kann vorkommen, dass so viele Fenster auf dem Bildschirm offen sind, dass ein Fenster komplett von anderen Fenstern überdeckt ist. Dann kann man offensichtlich nicht in das überdeckte Fenster klicken. Um das Fenster dennoch zu aktivieren, kann man durch Gedrückthalten der Alt-Taste und wiederholtes Drücken der Tab-Taste das zu fokussiernde Fenster auswählen. Ist man bei dem entsprechenden Fenster angelangt, so läßt man einfach die Alt-Taste los und schon springt das gewünschte Fenster in den Vordergrund.

Ebenso wie für Terminalfenster, gilt auch für alle anderen Fenster, die man während der Sitzung öffnet, dass sich alle Tastatur- und Mauseingaben immer auf das aktuelle Fenster beziehen.

Liste der virtuellen Bildschirme Hat man eine Vielzahl von Fenstern gleichzeitig geöffnet, so wird die Situation auf einem Bildschirm schnell unübersichtlich. Deswegen bietet der Fenstermanager insgesamt sechs sogenannte **virtuelle Bildschirme**. Jeder dieser Bildschirme wird im Prinzip wie ein eigener Monitor betrachtet. So lassen sich die Fenster über mehrere Bildschirme verteilen und man hat mehr Übersicht beim Arbeiten. Um zu einem anderen virtuellen Bildschirm zu wechseln muss man nur auf den entsprechenden Bildschirm in der Liste klicken.

Ist die Arbeit am Fenstermanager beendet, so muss man sich wieder ausloggen. Dies geschieht einfach, indem man mit der linken Taste auf eine freien Bereich klickt und in dem aufklappenden Menü das Kommando ,,Quit" wählt.

1.5.3 Navigieren im Dateisystem

Nachdem wir in den vorherigen Abschnitten gesehen haben, wie man sich am System anmeldet, wollen wir nun das Dateisystem erforschen.

Arbeitsverzeichnis

Unter LINUX ist jedem Programm, das gerade läuft, ein **Arbeitsverzeichnis** (*engl.* „working directory") zugeordnet. Dies gilt auch für die Shell, die beim Anmelden am System gestartet wird. Das Arbeitsverzeichnis für diese Shell ist das Home-Directory des Benutzers, der sich angemeldet hat. Um sich das aktuelle Arbeitsverzeichnis anzeigen zu lassen, benutzt man den Befehl `pwd` (**p**rint **w**orking **d**irectory):

```
fb04305:~ > pwd
/home/junglas
fb04305:~ >
```

Arbeitsverzeichnis wechseln

Das Arbeitsverzeichnis kann während der Arbeit am Rechner beliebig oft geändert werden (warum das sinnvoll ist, werden wir weiter unten sehen). Zu diesem Zweck benutzt man den Befehl `cd` (**c**hange **d**irectory):

```
fb04305:~ > pwd
/home/junglas
fb04305:~ > cd /opt
fb04305:~ > pwd
/opt
fb04305:~ > cd /usr
fb04305:~ > pwd
/usr
fb04305:~ > cd bin
fb04305:~ > pwd
/usr/bin
fb04305:~ > cd
fb04305:~ > pwd
/home/junglas
fb04305:~ >
```

Wie das Beispiel zeigt, macht der `cd`-Befehl das Verzeichnis zum Arbeitsverzeichnis, das dem Befehl als Parameter übergeben wird. Dabei gibt es zwei Besonderheiten:

1. Wird dem `cd`-Befehl kein Verzeichnis explizit angegeben, so wechselt der Befehl in das Heimatverzeichnis des Benutzers.

2. Das Zielverzeichnis muss nicht mit dem kompletten Pfadnamen angegeben werden (siehe `cd bin`): Beginnt der Name des Zielverzeichnisses nicht mit einem Schrägstrich (`/`), dann wird der Name des Arbeitsverzeichnisses vorangestellt. Im Beispiel wird beim Befehl `cd bin` also das Verzeichnis `/usr` vorangestellt und somit ins Verzeichnis `/usr/bin` gewechselt.

Der zweite Punkt zeigt bereits einen Vorteil des Arbeitsverzeichnisses: Datei- und Verzeichnisnamen können anstatt mit dem kompletten Pfadnamen auch mit einem Pfadnamen relativ zum Arbeitsverzeichnis angegeben werden. Im allgemeinen bezeichnet man Pfadnamen, die mit einem Schrägstrich beginnen, als **absolute Pfadnamen** (*engl.* „absolute pathnames") und solche, die nicht mit einem Schrägstrich beginnen als **relative Pfadnamen** (*engl.* „relative pathnames"). Tabelle 1.1 verdeutlicht diesen Sachverhalt noch einmal an einigen Beispielen (wobei vorausgesetzt wird, dass das aktuelle Verzeichnis `/home/junglas` ist).

relativ	entspricht absolut
`bin`	`/home/junglas/bin`
`data/Adressen.txt`	`/home/junglas/data/Adressen.txt`

Tabelle 1.1: Relative Pfade falls das Arbeitsverzeichnis `/home/junglas` ist.

Inhalte von Verzeichnissen anzeigen

Um sich die Dateien und Unterverzeichnisse eines Verzeichnisses anzeigen zu lassen, benutzt man das Programm `ls` (**l**i**s**t):

```
fb04305:~ > ls
CoMa.txt data
fb04305:~ > ls /
bin home www usr
fb04305:~ > ls /www
webpages homepages
fb04305:~ > ls /home/junglas
CoMa.txt data
```

`ls` listet alle Dateien und Verzeichnisse auf, die in der Argumentliste angegeben sind. Ist die Argumentliste leer, so wird angenommen, dass das Arbeitsverzeichnis aufgelistet werden soll. Ist in der Argumentliste ein Verzeichnis angegeben, so wird nicht das Verzeichnis selbst aufgelistet, sondern alle Dateien und Unterverzeichnisse, die sich in diesem Verzeichnis befinden.

Per Default werden Dateien, die mit einem Punkt beginnen, nicht aufgelistet. Möchte man diese Dateien auch sehen, so muss man die Option `-a` verwenden:

```
fb04305:~ > ls -a
. .. .bashrc CoMa.txt data
```

Mit dieser Option werden auch die beiden „speziellen" Einträge `.` und `..` aufgelistet. Diese beiden Einträge befinden sich in *jedem* Verzeichnis (also auch in leeren Verzeichnissen) und haben die folgende Bedeutung

`.` ist ein „Verweis" auf das Verzeichnis selber. So bezeichnen z.B. `/home/junglas`, `/home/junglas/.`, `/home/./junglas/./.` alle das gleiche Verzeichnis. Wir werden im Verlauf der Veranstaltung noch sehen, zu welchem Zweck man den „`.`"-Eintrag verwenden kann.

`..` ist ebenfalls eine Art Verweis, allerdings auf das übergeordnete Verzeichnis. Ist das aktuelle Verzeichnis `/home/junglas`, dann listet ,,`ls ..`" alle Dateien und Verzeichnisse in `/home` auf und ,,`cd ..`" wechselt in das Verzeichnis `/home`. Auch ,,`..`" kann in Pfadnamen vorkommen. So bezeichnen z.B. `/home/junglas`, `/home/../home/junglas` und `/home/junglas/../junglas` alle das gleiche Verzeichnis.

Im Wurzel-Verzeichnis `/` hat ,,`..`" eine etwas andere Bedeutung: da `/` keine übergeordnetes Verzeichnis hat, ist hier ,,`..`" ein Verweis auf `/`, bedeutet also das gleiche wie der ,,`.`"-Eintrag.

Verzeichnisse erstellen und löschen

Selbstverständlich kann man den Verzeichnisbaum auch dem eigenen Bedarf anpassen. Dazu muss man in der Lage sein, Verzeichnisse zu löschen oder neue Verzeichnisse zu erstellen. Zu diesem Zweck dienen die Programme `rmdir` (**rem**ove **dir**ectory) und `mkdir` (**m**a**k**e **dir**ectory). Dabei erstellt `mkdir` ein leeres Verzeichnis und `rmdir` löscht ein leeres Verzeichnis. Weitere Details zu diesen Programmen werden wir in den Übungen behandeln.

Übung: .....

1. mkdir,
2. rmdir vs. rm -rf,
3. Verzeichnisse . und .. (was heißt ,,leeres Verzeichnis"?)

.....

1.5.4 Dateien

Im vorigen Abschnitt haben wir erläutert, was man grundsätzlich für die Arbeit mit Verzeichnissen beherrschen/wissen muss. Viel interessanter als Verzeichnisse sind für den Benutzer allerdings Dateien, denn in ihnen werden nun tatsächlich Daten gespeichert (während Verzeichnisse nur dazu dienen, die Daten übersichtlich zu organisieren und damit – im Prinzip – auch weggelassen werden könnten).

Das LINUX-Betriebssystem betrachtet Dateien lediglich als eine Ansammlung von Bytes und unterscheidet nicht speziell zwischen Programmen und Daten. Demnach obliegt es also dem Benutzer zu entscheiden (oder noch besser: zu wissen) ob eine vorliegende Datei, ein Programm oder Daten enthält. Eine erste Hilfestellung hierzu liefert das Programm `file`. Dieses Programm entscheidet anhand der Struktur einer Datei oder des Dateiinhalts, um was für eine Datei es sich handelt:

```
fb04305:~ > file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), stripped
fb04305:~ > file data/Adressen.txt
data/Adressen.txt: ASCII text
fb04305:~ >
```

Im Beispiel zeigt `file`, dass `/bin/ls` eine ausführbare (executable) Datei, d.h. ein Programm ist, während es sich bei `data/Adressen.txt` um eine Text-Datei, d.h. um Daten handelt. Die restliche Ausgabe von `file` soll uns hier nicht weiter interessieren.

Hat man nun festgestellt, dass es sich bei einer vorliegenden Datei um eine Textdatei oder um sonst eine von Menschen lesbare (*engl.* „human readable") Datei handelt, kann man sich diese z.B. mit dem Programm `less` anzeigen lassen. Der Befehl

```
fb04305:~ > less data/Adressen.txt
```

zeigt den Inhalt der von `data/Adressen.txt` in dem Terminalfenster an. In diesem Text kann man durch bestimmte Buchstaben- oder Pfeiltasten der Tastatur hin und her springen. Tabelle 1.2 zeigt eine Auswahl der wichtigsten dieser Kommandos.

Taste	Wirkung
↓	bewegt den Text um eine Zeile nach unten
↑	bewegt den Text um eine Zeile nach oben
Bild ↓	bewegt den Text um eine Bildschirmseite nach unten
Bild ↑	bewegt den Text um eine Bildschirmseite nach oben
→	bewegt den Text nach rechts (falls es Textzeilen gibt, die nicht in einer Terminalzeile angezeigt werden können)
←	bewegt den Text nach links (falls es Textzeilen gibt, die nicht in einer Terminalzeile angezeigt werden können)
g	springt zum Anfang des Textes
G	springt zum Ende des Textes
q	beendet `less`

Tabelle 1.2: Kommandos für `less`.

Möchte man nun eine Datei nicht nur ansehen, sondern sie ändern, so braucht man dafür einen sogenannten „Editor". Wir werden für diese Veranstaltung hauptsächlich den Editor `emacs` oder dessen verbesserte (graphische) Variante `xemacs` verwenden. Wie man diese Editoren aufruft und benutzt werden wir in den Übungen lernen.

Übung: .....

1. less,
2. emacs,
3. xemacs,
4. xemacs braucht X11, emacs nur ncurses, d.h. emacs geht auch in der Textkonsole

.....

Nachdem nun klar ist, wie Dateien mit Hilfe eines Editors neu erstellt und verändert werden können, bleibt die Frage, wie man eine einmal erstellte Datei wieder los wird. Zu diesem Zweck gibt es das Programm `rm`. Der Befehl

```
fb04305:~ > rm data/Adressen.txt
fb04305:~ >
```

löscht z.B. die Datei `Adressen.txt` aus dem Unterverzeichnis `data` des Arbeitsverzeichnisses. Auch hier gilt wieder ,,no news is good news", d.h. solange das `rm`-Programm keine Fehlermeldung liefert, wurde die Datei erfolgreich gelöscht.

Achtung! Eine einmal gelöschte Datei kann unter LINUX nicht wieder hergestellt werden, unter keinen Umständen! Unter anderen Betriebssystemen ist das teilweise möglich.

1.5.5 Identität und Zugriffsrechte

Wie bereits erwähnt, ist LINUX ein Mehrbenutzerbetriebssystem. Jeder Benutzer hat gegenüber dem System eine Identität. Diese Identität beinhaltet einen Benutzernamen und damit eindeutig verknüpft eine **Benutzernummer**, die sogenannte **User-ID**. Zusätzlich ist jeder Benutzer noch Mitglied in mindestens einer **Gruppe**, der sogenannten **Primary Group** (oder kurz **Group**). Auch diese Gruppe hat einen Namen und eine eindeutige Nummer, die sogenannte **Group-ID**. Um festzustellen, als welcher Benutzer man angemeldet ist, kann man das Programm `id` verwenden:

```
fb04305:~ > id
uid=713(junglas) gid=27(ag07) groups=27(ag07)
```

Dieses Beispiel zeigt, dass man aktuell als Benutzer ,,junglas" mit der User-ID 713 eingeloggt ist. Die Gruppe ist ,,ag07" und hat die Nummer 27. Ausser in Gruppe ,,ag07" ist ,,junglas" sonst in keiner anderen Gruppe.

Ein wesentliches Charakteristikum von Mehrbenutzersystemen ist, dass man bei selbst erstellten Daten/Dateien entscheiden kann, wer was damit ,,machen" kann und man mit fremden Daten/Dateien nur das tun darf, was einem der Besitzer erlaubt hat.

Generell kann man unter LINUX drei Dinge mit Dateien ,,machen"

1. Dateien können gelesen werden,
2. Dateien können geschrieben (d.h. verändert oder gelöscht) werden und
3. Dateien können ausgeführt werden.

All dies kann man anderen Leuten für seine eigenen Dateien erlauben oder verbieten. Dabei kann man die jeweilige Erlaubnis (oder das Verbot) jeweils für die folgenden Mengen an Benutzern erteilen

1. man selbst (user),
2. Benutzer aus der gleichen Gruppe (group),
3. beliebige Benutzer (other).

Nur wer explizit die Erlaubnis hat, eine Datei zu lesen, zu schreiben oder auszuführen kann dies auch tun.

Betrachten wir einmal folgendes Beispiel: Der Befehl

```
fb04305:~ > touch Datei
```

erzeugt eine leere Datei names `Datei` im Arbeitsverzeichnis. Für diese Datei sind nun die Default-zugriffsrechte gesetzt, die wir uns mit `ls -l` ansehen können:

```
fb04305:~ > ls -l Datei
-rw-r--r-- 1 junglas ag07 ... Datei
```

Wir sehen damit zunächst, dass die Datei dem Benutzer ,,junglas" aus der Gruppe ,,ag07" gehört. Ganz links finden wir die Zeichenkette `-rw-r--r--`. Das erste '-'-Zeichen ignorieren wir, sodass `rw-r--r--` übrig bleibt. Das sind neun Zeichen, die in jeweils drei Dreiergruppen zu interpretieren sind. Dabei spezifiziert das erste Triple, was der Benutzer mit der Datei machen darf, das zweite, was Mitglieder der gleichen Gruppe mit der Datei machen dürfen und das dritte, was beliebige Benutzer mit der Datei machen dürfen.

Übung: .....

1. Was bedeutet das erste '-', bzw. d/l/c/b/s?

.....

Jedes dieser Tripel besteht aus den Buchstaben 'r', 'w' und 'x' bzw. aus einem '-' an der entsprechenden Stelle. Ein 'r' (read) erlaubt der jeweiligen Benutzermenge das Lesen der Datei, ein 'w' (write) das Schreiben und ein 'x' (execute) das Ausführen. Ein '-'-Zeichen an einer Stelle verbietet die entsprechende Aktion.

Die Menge der erlaubten und verbotenen Aktionen für eine Datei bezeichnen wir als die **Zugriffsrechte** (*engl.* ,,permissions" oder *engl.* ,,mode") der Datei. Abkürzend spricht man anstatt von Zugriffsrechten häufig einfach auch von ,,Rechten".

Um die Rechte einer Datei zu ändern benutzt man das Programm `chmod`, das wir in der Übung näher betrachten werden.

Übung: .....

1. chmod

.....

1.5.6 Hilfe

Wir haben in den vorherigen Abschnitten bereits einige der am häufigsten verwendeten Programme vorgestellt. Zu einigen Programmen haben wir ausserdem eine (unvollständige) Liste an Argumenten und/oder Parametern angegeben. Unter `/bin`, `/usr/bin` bzw. `/usr/local/bin` finden sich aber weitaus mehr Programme, als wir bisher vorgestellt haben. Wie findet man heraus, was diese Programme tun und wie sie aufgerufen werden?

Unter Linux gibt es zu diesem Zweck die sogenannten ,,Manual-" und ,,Info-Seiten". Auf diesen Seiten wird (meist sehr ausführlich) beschrieben, wozu ein Programm dient und wie es aufgerufen werden kann. Die Manual-Seite zum `rm`-Programm findet man durch den Befehl

```
fb04305:~ > man rm
```

Die zugehörige Manual-Seite wird dann ähnlich wie mit dem `less`-Programm (siehe Abschnitt 1.5.4) im Terminalfenster angezeigt. Eine Manual-Seite zu einem Programm gliedert sich normalerweise in mehere Sektionen, von denen die folgenden die wichtigsten sind:

Name Hier steht der der Name und eine sehr kurze Beschreibung des Programms.

Synopsis In dieser Sektion werden die verschiedenen Möglichkeiten aufgelistet, wie das Programm aufgerufen werden kann.

Description Dieser Teil der Manual-Seite liefert eine ausführliche Beschreibung des Programms und seiner Wirkungsweise.

Options Die Optionen, die in der ,,Synopsis"-Sektion bereits aufgelistet wurden, werden hier detailliert vorgestellt.

Eine genauere Beschreibung des `man`-Programms liefert – wie sollte es anders sein – der Befehl

```
fb04305:~ > man man
```

Obwohl die Manual-Seite eines Programms meistens dazu ausreicht, Funktionsweise und Benutzung des Programms zu verstehen, gibt es für die meisten Programme auch eine Info-Seite. Für das `rm`-Programm ruft man diese mittels

```
fb04305:~ > info rm
```

auf. Die Info-Seiten zu einem Programm sind meistens ausführlicher und anders/besser gegliedert als die Manual-Seiten. Ausserdem finden sich auf den Info-Seiten häufig auch Beispiele, die es einem leichter machen, das Programm zu benutzen, insbesondere, wenn man dies zum ersten Mal tut. Mehr Informationen über das Ansehen der Info-Seiten sowie eine kurze Einführung erhält man durch

```
fb04305:~ > info info
```

Weiss man dagegen noch überhaupt nicht, welches Programm man überhaupt benutzen soll, so hilft einem das Programm `apropos` weiter. Angenommen wir wollen die Zugriffsrechte auf eine Datei ändern, wissen aber nicht, welches Programm wir dafür verwenden sollen/müssen. Im folgenden Beispiel haben wir das Programm `apropos` benutzt, um das herauszufinden:

```
fb04305:~ > apropos permission
access (2) - check user's permissions for a file
check_perms (8) - Check permissions of Mailman's files
chmod (1) - change file access permissions
chmod (2) - change permissions of a file
fchmod (2) - change permissions of a file
ioperm (2) - set port input/output permissions
mysql_setpermission (1) - No manpage for this program, utility or function.
fb04305:~ > man chmod
```

`apropos` liefert zunächst eine ganze Liste von Programmen, die etwas mit dem Thema ,,permissions" zu tun haben. Aus dieser Liste mußten wir jetzt noch das richtige ausfiltern (im schlimmsten Fall hätten wir einfach alle aufgelisteten Programme ausprobiert). Anhand der Kurzbeschreibungen auf der rechten Seite war aber sofort klar, dass wir das Program `chmod` brauchen.

1.5.7 Umgebungsvariablen

Jeder Prozess unter LINUX verfügt über eine sogenannte **Umgebung** (*engl.* ,,Environment"). In dieser Umgebung sind Einstellungen gespeichert, die das System an die Bedürfnisse des jeweiligen Benutzers anpassen. Man kann sich diese Umgebung mit dem Programm `env` anzeigen lassen:

```
fb04305:~ > env
HOME=/home/junglas
USER=junglas
PAGER=less
LS_COLORS=no=00:fi=00:di=01;34:ln=00;36:pi=40;33:so=01;35:do=01;35:bd=40;33;01:
cd=40;33;01:or=40;31:ex=00;32:*.cmd=01;32:*.exe=01;32:*.com=01;32:*.bat=01;32:
*.btm=01;32:*.dll=01;32:*.tar=00;31:*.tbz=00;31:*.tgz=00;31:*.rpm=00;31:
*.deb=00;31:*.arj=00;31:*.taz=00;31:*.lzh=00;31:*.zip=00;31:*.zoo=00;31:
*.z=00;31:*.Z=00;31:*.gz=00;31:*.bz2=00;31:*.tb2=00;31:*.tz2=00;31:
*.tbz2=00;31:*.avi=01;35:*.bmp=01;35:*.fli=01;35:*.gif=01;35:*.jpg=01;35:
*.jpeg=01;35:*.mng=01;35:*.mov=01;35:*.mpg=01;35:*.pcx=01;35:*.pbm=01;35:
*.pgm=01;35:*.png=01;35:*.ppm=01;35:*.tga=01;35:*.tif=01;35:*.xbm=01;35:
*.xpm=01;35:*.dl=01;35:*.gl=01;35:*.aiff=00;32:*.au=00;32:*.mid=00;32:
*.mp3=00;32:*.ogg=00;32:*.voc=00;32:*.wav=00;32:
LS_OPTIONS=-N --color=tty -T 0
PATH=/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin
INFOPATH=/usr/local/info:/usr/share/info:/usr/info
MANPATH=/usr/local/man:/usr/share/man:/usr/X11R6/man
...
fb04305:~ >
```

Wie das Beispiel zeigt, besteht die Umgebung eines Prozesses aus mehreren Name/Wert-Paaren. Den Namen ist jeweils durch ein Gleichheitszeichen ein Wert zugewiesen. Diese Namen werden auch als **Umgebungsvariablen** (*engl.* ,,Environment Variables") bezeichnet. Der Name einer solchen Variablen beginnt mit einem Unterstrich oder einem Buchstaben (kein Umlaut!) gefolgt von einer beliebigen Anzahl von Zahlen, Unterstrichen und Buchstaben (wieder keine Umlaute!).

Einige der wichtigsten Umgebungsvariablen aus dem Beispiel von oben sind

HOME Diese Variable enthält den Namen des Heimatverzeichnisses des Benutzers.

USER Der Login-Name des Benutzers.

PAGER Viele Programme wie z.B. `man` zeigen lange Textseiten an, die üblicherweise nicht auf einmal in einem Terminalfenster dargestellt werden können. Damit die Texte dennoch sinnvoll gelesen werden können, werden sogenannte ,,Pager" verwendet, die immer nur soviel vom Text anzeigen, wie in das Terminalfenster paßt. Die Pager bieten dann meist die Möglichkeit, den Text im Terminalfenster nach oben oder nach unten zu scrollen. Einer dieser Pager ist das Programm `less`, das wir bereits im Abschnitt 1.5.4 kennengelernt haben. Weitere Pager sind z.B. `more` oder `pg`.

Die Variable `PAGER` gibt an, welchen Pager die Programme wie `man` verwenden sollen, sofern nicht explizit ein anderer Pager angegeben wird.

LS_COLORS, LS_OPTIONS Das Programm `ls` benutzt die Werte dieser Variablen um die Liste der angezeigten Dateien zu formatieren. Die Werte dieser beiden Variablen im Beispiel bewirken, dass Dateien je nach ihrem Typ in einer anderen Farbe angezeigt werden.

PATH Unter LINUX sind ausführbare Programme wie `ls`, `mkdir`, `man` usw. an verschiedenen Plätzen im Dateibaum installiert. Damit man nicht immer langwierig den absoluten Dateinamen dieser Programme eingeben muss, benutzt man die `PATH`-Variable: Diese Variable enthält eine Liste von Verzeichnissen, die jeweils durch einen Doppelpunkt getrennt sind (demnach sollten Doppelpunkte möglichst nicht in Verzeichnisnamen vorkommen!). Gibt man dann in der Shell z.B.

```
fb04305:~ > my_prog
```

ein, dann werden alle in `PATH` aufgelisteten Verzeichnisse in der angegebenen Reihenfolge nach einem Programm mit Namen `my_prog` durchsucht. Wird so ein Programm gefunden, dann wird es ausgeführt, anderenfalls antwortet die Shell mit einer Fehlermeldung:

```
fb04305:~ > my_prog
sh: my_prog: command not found
fb04305:~ >
```

Im Normalfall ist `PATH` beim Starten der Shell so vorbesetzt, dass alle vorinstallierten Programme ohne Pfadangabe gestartet werden können. Eine Änderung der `PATH`-Variablen wird meist erst dann nötig, wenn man selbst neue Programme schreibt oder installiert.

INFOPATH Die Info-Seiten sind je nach Programm an verschiedenen Stellen des Dateisystems installiert. Die `INFOPATH`-Variable spezifiert (wie die `PATH`-Variable) eine Liste von Verzeichnissen, in denen nach Info-Seiten gesucht wird, sobald die Info-Seite für ein bestimmtes Programm angefordert wird.

MANPATH Wie die Info-Seiten sind auch die Manual-Seiten an verschiedenen Stellen im Dateisystem installiert. Die `MANPATH`-Variable gibt die Liste von Verzeichnissen an, die durchsucht werden sollen, wenn nach einer bestimmten Manual-Seite gesucht wird.

Environment-Variablen auslesen und benutzen

Mit dem Programm `env` haben wir bereits eine Möglichkeit kennengelernt, wie man sich die Werte *aller* Environment-Variablen anzeigen lassen kann. Im Normalfall ist man aber nicht an allen, sondern nur an einer oder kleinen Auswahl interessiert. Um auf einzelne Variablen zugreifen zu können, bedient man sich des Dollar-Zeichens:

```
fb04305:~ > ls $LS_OPTIONS
```

Trifft die Shell beim Einlesen eines Befehls auf ein Dollar-Zeichen, so werden alle Zeichen bis zum nächsten Leerzeichen, Dollar-Zeichen, Tabulator oder Zeilenende als Variablenname betrachtet. Anstelle des Dollar-Zeichens und des Variablennamens wird dann *vor* der Ausführung des Programms der Wert der entsprechenden Variablen eingesetzt. Ist also `LS_OPTIONS` wie oben auf ,,`-N --color=tty -T 0`" gesetzt, dann sind

```
fb04305:~ > ls -N --color=tty -T 0
```

und

```
fb04305:~ > ls $LS_OPTIONS
```

äquivalent. Um einfach den Wert einer Environment-Variablen auszugeben, bedient man sich des `echo`-Befehls:

```
fb04305:~ > echo $LS_OPTIONS
-N --color=tty -T 0
```

Der `echo`-Befehl gibt allen Text aus, der dem Wort ,,echo" in der Kommandozeile folgt. In unserem Fall wird ,,`$LS_OPTIONS`" durch den Wert der Variablen `LS_OPTIONS` (nämlich ,,`-N --color=tty -T 0`") ersetzt, bevor der `echo`-Befehl ausgeführt wird.

Achtung! Das Dollar-Zeichen ist *nicht* Bestandteil des Variablennamens. Es dient lediglich dazu, der Shell anzuzeigen, dass der folgende Text ein Variablenname ist und der Inhalt dieser Variablen eingesetzt werden soll.

Environment-Variablen setzen und definieren

Am Beispiel der Variablen `LS_COLORS` haben wir gesehen, wie man Environment-Variablen benutzen kann, um häufig benutzte Befehle oder Textsequenzen abzukürzen (bzw. wie diese Abkürzungen von Programmen wie `ls` implizit verwendet werden). Zu diesem Zweck muss man sich diese Variablen allerdings erst einmal definieren.

Wie bereits erwähnt, wird am LINUX-Pool des Fachbereichs Mathematik üblicherweise die `bash` verwendet. In dieser Shell setzt man eine Environment-Variable mittels

```
fb04305:~ > export Name=Wert
```

Dabei ist ,,`Name`" der Name der Variablen, die gesetzt werden soll. Existiert bereits eine Variable mit diesem Name, so wird ihr Wert überschrieben. Andernfalls wird eine neue Variable mit diesem Namen erstellt.

,,`Wert`" ist der Wert, der der Variablen `Name` zugewiesen werden soll. Die folgende Zeile erzeugt demnach eine Variable `home_dir` mit dem Wert `/home/junglas`.

```
fb04305:~ > export home_dir=/home/junglas
```

Hat eine Variable `home_dir` schon vorher existiert, so wird die alte ,,Version" überschrieben.

Die Datei `.bashrc` im Heimatverzeichnis

Environment-Variablen wie `PATH`, `MANPATH` oder `LS_OPTIONS` werden sehr häufig verwendet. Daher wäre es sehr mühsam, wenn man jedesmal, nachdem man eine Shell gestartet hat, erst einmal alle diese Variablen per Hand aufsetzen müßte. Um diese Arbeit einzusparen kann man die Datei `.bashrc` im Heimatverzeichnis benutzen. Die `bash` liest beim Start diese Datei und führt alle Anweisungen in ihr aus. Schreibt man also in diese Datei eine Zeile der Art

```
export LS_OPTIONS='-N --color=tty -T 0'
```

dann wird bei jedem Start einer Shell die Environment-Variable `LS_OPTIONS` automatisch auf den Wert ,,`-N --color=tty -T 0`" gesetzt. Da sich das Environment einer Shell auf alle Prozesse vererbt, die aus dieser Shell gestartet werden, ist `LS_OPTIONS` also für alle Programme gesetzt.

Dabei ist zu beachten, dass einige Variablen wie `HOME`, `PWD` oder `OLDPWD` von der Shell automatisch gesetzt werden. Zuweisungen an diese Variablen sind zwar möglich, sollten aber tunlichst unterlassen werden.

Quoting

Soll der Wert einer Umgebungsvariablen Leerzeichen, Tabulatoren oder andere sogenannte ,,Whitespaces"enthalten, möchte man eine Datei erstellen, deren Name diese Zeichen enthält, oder enthält einer der Parameter für ein Programm diese Zeichen, dann steht man vor dem Problem, dass die Shell diese Whitespaces zur Trennung syntaktischer Einheiten benutzt. Möchte man z.B. erreichen, dass die Variable `LS_COLORS` wie oben besetzt wird, dann kann man dies nicht durch

```
fb04305:~ > export LS_COLORS=-N --color=tty -T 0
```

erreichen. Die Shell liest in diesem Fall nämlich nur `LS_COLORS=-N` und betrachtet den Rest der Zeile als eine Liste von Variablennamen. Da dies aber keine gültigen Variablennamen sind, führt obige Zeile zu einem Fehler.

Wenn man also einen Wert zuweisen möchte, der Whitespaces enthält, so muss man diese ,,quoten". Dafür gibt es insgesamt drei Möglichkeiten:

```
fb04305:~ > export LS_OPTIONS="-N --color=tty -T 0"
fb04305:~ > export LS_OPTIONS='-N --color=tty -T 0'
fb04305:~ > export LS_OPTIONS=-N\ --color=tty\ -T\ 0
```

D.h. entweder man schließt den Wert in doppelte oder einfache Anführungszeichen ein, oder man schreibt vor jeden Whitespace (in unserem Fall sind das nur Leerzeichen) einen **Rückstrich** (*engl.* ,,Backslash"). Der Unterschied zwischen doppelten und einfachen Anführungszeichen ist etwas subtil und wir werden diesen in den Übungen behandeln.

Übung: .....

1. Substitution in doppelten Anführungszeichen,
2. keine Substitution in einfachen Anführungszeichen,
3. Backslash für $

.....

Ähnliche Probleme wie beim zuweisen von Werten, die Whitespaces enthalten, treten auch auf, wenn man versucht mit Dateien zu arbeiten, die z.B. Leer- oder Dollarzeichen enthalten, wie dies unter WINDOWS© üblich ist. Zwar lassen sich die Probleme hier auch durch Quoting lösen, trotzdem sollte man solche Dateinamen möglichst vermeiden.

1.5.8 Wildcards

Mit den Environment-Variablen aus Abschnitt 1.5.7 haben wir bereits eine Möglichkeit gefunden, wie man häufig verwendete Befehle oder Textsequenzen abkürzen kann und sich somit die Arbeit vereinfachen kann. Eine weitere Möglichkeit sind sogenannte **Wildcards** (*deut.* „Jokerzeichen"). In diesem Abschnitt wollen wir nun die Arbeit mit diesen Zeichen beschreiben.

Unter der am LINUX-Pool des Fachbereich Mathematik verwendeten Shell (`bash`) gibt es zwei Wildcards: den Stern ('*') und das Fragezeichen ('?'). Dabei steht der Stern für eine *beliebige Anzahl* beliebiger Zeichen und das Fragezeichen für *ein* beliebiges Zeichen im Dateinamen. Die Arbeitsweise der Wildcards läßt sich am leichtesten durch ein Beispiel veranschaulichen:

Angenommen, das Arbeitsverzeichnis enthält die Dateien `file_`, `file_1`, `file_2` und `file_3`, d.h.

```
fb04305:~ > ls
file_ file_1 file_2 file_3
```

Möchten wir nun alle Dateien ausser der Datei `file_` löschen, können wir dies durch eine der beiden folgenden Zeilen erreichen:

```
fb04305:~ > rm file_1 file_2 file_3
fb04305:~ > rm file_?
```

Die erste Form haben wir bereits kennengelernt, die zweite Form benutzt eine Wildcard. Beim Einlesen des Befehls ersetzt die Shell den Text `file_?` durch alle Dateinamen, die aus `file_` gefolgt von *einem* beliebigen Zeichen bestehen. In unserem Beispiel sind das gerade `file_1`, `file_2` und `file_3`. Dementsprechend würde

```
fb04305:~ > rm ?
```

alle Dateien im Arbeitsverzeichnis löschen, deren Name aus nur einem Buchstaben bestehen.

Im Vergleich zu

```
fb04305:~ > rm file_?
```

dagegen würde

```
fb04305:~ > rm file_*
```

alle Dateien im Arbeitsverzeichnis löschen, deren Name aus `file_` gefolgt von einer *beliebigen Anzahl* beliebiger Zeichen besteht. Dies würde also auch die Datei `file_` betreffen, genauso aber auch eventuell existierende Dateien `file_123` oder `file_abc`. Analog würde der Befehl

```
fb04305:~ > rm *
```

alle Dateien aus dem Arbeitsverzeichnis entfernen.

Überall wo eine Liste von Dateien verlangt wird, kann man diese Liste mit Hilfe der Wildcards abkürzen bzw. erzeugen (sofern die aufzulistenden Dateinamen hinreichend ähnlich aussehen). Dabei müssen die Wildcards nicht am Ende des jeweiligen Musters stehen, sondern können sich irgendwo im Muster befinden. Tabelle 1.3 zeigt einige Beispiele für Muster, die mit Wildcards erstellt werden können.

Muster	paßt auf
a*	alle Dateinamen im Arbeitsverzeichnis, die mit 'a' anfangen
*a	alle Dateinamen im Arbeitsverzeichnis, die mit 'a' aufhören
a*b	alle Dateinamen im Arbeitsverzeichnis, die mit 'a' anfangen und mit 'b' aufhören
a?b	alle Dateinamen im Arbeitsverzeichnis, die aus drei Zeichen bestehen und mit 'a' anfangen und mit 'b' aufhören
a*b?c	alle Dateinamen im Arbeitsverzeichnis, die mit 'a' anfangen, deren drittletzter Buchstabe ein 'b' ist und die mit 'c' aufhören
a	alle Dateinamen im Arbeitsverzeichnis, die ein 'a' enthalten
/bin/l*	alle Dateinamen im Verzeichnis `/bin`, die mit 'l' beginnen
/*i*/*	alle Dateinamen, die in einem Verzeichnis unter `/` liegen, dessen Name ein 'i' enthält

Tabelle 1.3: Dateimuster mit Wildcards.

Übung: .....

1. Jobs im Hintergrund mit '&'.
2. Drucken (`lpr`, `a2ps`, `gv`, `acroread`).

.....

Kapitel 2

Die Programmiersprache `C`

The best book on programming for the layman is Alice in Wonderland*,*
but that's because it's the best book on anything for the layman.
— Alan J. Perlis

Nachdem wir uns in Kapitel 1 mit dem System vertraut gemacht haben, auf dem wir in Zukunft arbeiten, wollen wir nun mit der Beschreibung der Programmiersprache beschäftigen, die wir für den Rest der Vorlesung verwenden.

Eine solche Programmiersprache wird benötigt, wenn man nicht nur die Programme benutzen will, die auf dem Rechner bereits installiert sind oder im Internet angeboten werden, sondern eigene Programme schreiben möchte, um bestimmte Probleme zu lösen.

Wir haben uns dabei für die Programmiersprache `C` entschieden, denn

- `C` ist unter Linux die wohl am meisten verbreitete Programmiersprache. Nahezu das komplette Betriebssystem ist in `C` programmiert.
- Die beiden beliebten objektorientierten Programmiersprachen `C++` und `Java` bauen in vielen Teilen auf `C` auf.
- Mit Hilfe von `C` lassen sich bereits fast alle Probleme lösen, auf die man im mathematischen Alltag trifft.
- Für die Programmierung in `C` gibt es eine Vielzahl von Hilfsprogrammen, die einem das Programmieren deutlich erleichtern.
- In `C` lassen sich viele Dinge effizienter implementieren als in anderen Programmiersprachen – auch wenn der Aufwand dadurch ab und zu grösser wird.

Wir wollen die Programmiersprache `C` zunächst an einigen sehr einfachen Beispielen kurz vorstellen und dann näher auf das tatsächliche Programmieren eingehen.

Es sei allerdings bereits jetzt darauf hingewiesen, dass das formale Erlernen einer Programmiersprache lediglich die Grundlage für das Programmieren bildet und man gutes Programmieren

selbst nur durch ständiges Üben lernen kann. Während man eine Programmiersprache meist innerhalb einer Woche oder eines Monats lernen kann, so braucht das Erlernen guter und erfolgreicher Programmiertechniken meist mehrere Monate oder Jahre.

2.1 Einführende Beispiele

In diesem Abschnitt geben wir zunächst zwei einfache Beispiele von C-Programmen, an denen sich bereits die meisten Bestandteile von solchen Programmen sehr einfach erkennen lassen. Allgemein ist ein C-Programm eine Liste von Anweisungen, die beim Ausführen des Programmes nacheinander ausgeführt werden.

2.1.1 „Hello World!"

Das „Hello World!"-Programm ist das wohl bekannteste Beispielprogramm. Es gibt kaum ein Buch über eine Programmiersprache, in dem nicht als erstes gezeigt wird, wie man „Hello World!" auf der Konsole ausgibt.

In C schreibt man zu diesem Zwecke eine Datei `hello.c` mit folgendem Inhalt:

```
#include <stdio.h>

int main (void)
{
  printf("Hello world!\n");
  return 0;
}
```

Danach gibt man in der Shell

```
$ gcc -W -Wall -o hello hello.c
```

ein. `gcc` ist ein sogenannter **Compiler**, der Programmcode in einer „Hochsprache" (in diesem Fall C) in Maschinencode übersetzt. Nach der Übersetzung durch den Compiler kann das Programm dann genauso ausgeführt werden, wie wir das bisher bei anderen Programmen gesehen haben:

```
$ ./hello
```

Auf der Konsole sollte nun der Text „Hello World!" erscheinen.

Das „Hello World!"-Programm zeigt bereits einige der wichtigsten Bestandteile eines C-Programms:

- Präprozessor-Anweisungen (Zeilen, die mit `#` beginnen),
- Funktionsdefinitionen (`main`),
- Funktionsaufrufe (`printf(...)`),
- Blöcke (Text (=Code) der zwischen geschweiften Klammern steht),

- Rückgabewerte (`return`) von Funktionen.

Außerdem erfüllt das „Hello-World!"-Programm die Anforderung, die an jedes C-Programm gestellt wird: Es muß *genau eine* Funktion namens `main` definiert sein. Diese Funktion wird dann beim Aufruf des Programms ausgeführt.

2.1.2 Fibonacci-Zahlen

Ein weiteres beliebtes Beispielprogramm ist die Ausgabe von Fibonacci-Zahlen. Die Fibonacci-Zahlen sind eine Folge $(F_n)_{n\in\mathbb{N}}$ von Zahlen, die wie folgt definiert sind

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \quad \text{für } n \geq 2. \end{aligned}$$

Möchte man den Anfang dieser Zahlenfolge auf der Konsole ausgeben, dann kann das mit folgendem C-Programm erreicht werden:

```
#include <stdio.h>

/*
 * File fib.c:
 * Print the first few numbers of the Fibonacci sequence.
 */

int main (void)
{
  int low, high;  /* declare integer variables "low" and "high" */
  int sum = 0;    /* declare AND initialise integer variable "sum" */

  /* first Fibonacci number */
  low = 0;
  printf("%d\n", low);

  /* second Fibonacci number */
  high = 1;
  printf("%d\n", high);

  /* subsequent Fibonacci numbers */
  while (sum < 50) {
    sum = high + low;
    printf("%d\n", sum);
    low = high;
    high = sum;
  }
```

```
  return 0;
}
```

Dieses Programm wird analog zum ,,Hello World!"-Programm in eine Datei namens `fib.c` geschrieben, mit `gcc -W -Wall -o fib fib.c` übersetzt und mit `./fib` ausgeführt.

Das Fibonacci-Programm zeigt einige weitere interessante Bestandteile von C-Programmen:

- Kommentare (`/* ... */`),
- Variablen-Definitionen (`int low`),
- Zuweisungen (`low = 0`),
- Kontrollblöcke (`while (...) { ... }`),
- arithmetische Operationen (`high + low`),
- formatierte Ausgabe mit `printf`.

Diese Bestandteile sowie diejenigen, die bereits im ,,Hello World!"-Programm auftraten, werden wir in diesem Kapitel genauer unter die Lupe nehmen. Vorher müssen wir allerdings noch einige formale Definitionen klären:

2.2 Syntax und Semantik

Bei jedem Computerprogramm unterscheiden wir Syntax und Semantik:

- Die **Syntax** sind die Regeln, nach denen der Programmcode geschrieben wird. Jeder Compiler definiert seine eigene Syntax, sodass Programme in verschiedenen Programmiersprachen verschieden aussehen. Das ,,Fibonacci"-Programm würde in der Programmiersprache `Perl` z.B. so aussehen:

```
#!/usr/bin/perl -w

$low = 0;
print "$low\n";

$high = 1;
print "$high\n";

$sum = 0;
while ($sum < 50) {
  $sum = $high + $low;
  print "$sum\n";
  $low = $high;
  $high = $sum;
}

exit(0);
```

Die beiden Programme erzeugen exakt die gleichen Ausgaben, sind aber in unterschiedlichen Programmiersprachen implementiert.

- Die **Semantik** eines Programms ist das (abstrakte) Etwas, was das Programm tut. Die Semantik des ,,Hello World!"-Programm ist z.B. die Ausgabe eben dieses Textes auf der Konsole. Insbesondere hängt die Semantik eines Programms also nicht von der Programmiersprache ab, in der das Programm geschrieben wurde.

Damit ein Programm von einem `C`-Compiler gelesen und übersetzt werden kann, muss es konform mit der `C`-Syntax sein.

Das Einlesen einer Datei mit `C`-Code erfolgt durch einen sogenannten **Parser**. Dieser Parser betrachtet die Eingabedatei als eine Reihe von **Token**, die durch **Whitespaces** voneinander getrennt sind. Ein Whitespace ist z.B. ein Leerzeichen, ein Zeilenvorschub oder ein Tabulator. Dabei ist es egal, wieviele Whitespaces benutzt werden, um zwei Token zu trennen, Voraussetzung ist nur, dass es mindestens ein Whitespace ist.

Eine Ausnahme bilden Zeichenketten, die in doppelte Anführungszeichen eingeschlossen sind: Text zwischen Paaren von doppelten Anführungszeichen wird immer als *ein* Token betrachtet. Enthält der eingeschlossene Text Whitespaces, so werden diese als Teil des Tokens betrachtet.

Einige spezielle Zeichen wie z.B. Klammern ('(', ')', '[', ']', '{' und '}') werden – solange sie nicht in doppelte Anführungszeichen eingeschlossen sind – immer als ein Token betrachtet und müssen deshalb nicht durch Whitespaces von anderen Token getrennt werden

Die Tokens im ,,Hello World!"-Beispiel von oben sind: `#include`, `<stdio.h>`, `int`, `main`, `(`, `void`, `)`, {, `printf`, `(`, `"Hello world!\n"`, `)`, `;`, `return`, `0` und }.

Nachdem der Parser die Eingabedatei in Tokens zerlegt hat, wird überprüft, ob die Tokens die Syntaxregeln für `C` erfüllen. Ist dies nicht der Fall, so wird ein ,,Syntax Error" ausgegeben und die Bearbeitung der Eingabedatei wird abgebrochen. In diesem Fall muss der Programmierer den Syntaxfehler beheben und die Eingabedatei erneut übersetzen.

2.3 Variablen

Variablen dienen einem Programm dazu, Informationen, die während der Laufzeit des Programms benötigt werden, im Hauptspeicher abzulegen. Grundsätzlich könnten diese Informationen auch temporär auf der Festplatte gespeichert werden. In Abschnitt 1.1 haben wir aber bereits darauf hingewiesen, dass Zugriffe auf die Festplatte deutlich langsamer vonstatten gehen als Zugriffe auf den Hauptspeicher.

In `C` hat jede Variable einen Namen, der aus einem oder mehreren Zeichen besteht. Dabei muss das erste Zeichen ein Buchstabe oder ein Unterstrich sein, alle weiteren Zeichen dürfen Zahlen, Buchstaben oder Unterstriche sein. Buchstaben sind in diesem Sinne die Kleinbuchstaben 'a' bis 'z' und die Großbuchstaben 'A' bis 'Z'. Insbesondere sind also Umlaute wie 'ä', 'Ö' oder 'ß' in Variablennamen nicht erlaubt.

Ausserdem gibt es in `C` 32 sogenannte **Schlüsselwörter** (*engl.* ,,keywords"), von denen wir bereits einige gesehen haben. Diese Schlüsselwörter haben spezielle Bedeutungen und dürfen deshalb nicht als Variablennamen verwendet werden. Tabelle 2.1 listet alle 32 Schlüsselwörter von `C` auf.

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

Tabelle 2.1: Schlüsselwörter in C.

2.3.1 Typen

Variablen reservieren einen gewissen Platz im Hauptspeicher des Computers. Dieser Platz ist nach der Reservierung ausschließlich dem Prozeß (d.h. dem Programm) vorbehalten, der diesen Platz reserviert hat, d.h. Prozesse können keine Speicherbereiche beschreiben, die von einem anderen Prozeß bereits reserviert worden sind. Da sich der Platzbedarf für die zu speichernde Information nach der Größe der Information richtet, gibt es verschiedene Variablen, die jeweils eine unterschiedliche Anzahl an Bits reservieren und die Werte der Bits unterschiedliche interpretieren. Der Programmierer selbst braucht sich um diese Interpretation im Grunde nicht zu kümmern. Er sollte lediglich grob wissen, wie sie funktioniert. Ansonsten kann er einfach den Variablen, die er benutzt, entsprechende Werte zuweisen oder die Werte aus den Variablen auslesen, ohne sich darum zu kümmern, wie diese Werte nun tatsächlich im Speicher abgelegt werden. Tabelle 2.2 gibt einen Überblick über die verschiedenen Variablentypen, die in C zur Verfügung stehen. Dabei

	Vorzeichen	kleinster Wert	größter Wert	Größe in Bit
`char`	ja	-128	127	8
`unsigned char`	nein	0	255	8
`short`	ja	-32768	32767	16
`unsigned short`	nein	0	65535	16
`long`	ja	-2147483648	2147483647	32
`unsigned long`	nein	0	4294967295	32
`float`	ja	$1.2 \cdot 10^{-28}$	$3.4 \cdot 10^{38}$	32
`double`	ja	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$	64
`int`	ja	?	?	?
`unsigned int`	nein	?	?	?
`size_t`	nein	?	?	?
`ssize_t`	ja	?	?	?
`off_t`	nein	?	?	?
`long long`	ja	$-(2^{63})$	$2^{63}-1$	64
`unsigned long long`	nein	0	$2^{64}-1$	64
`long double`	ja	$3.4 \cdot 10^{-4932}$	$1.2 \cdot 10^{4932}$	96

Tabelle 2.2: Variablentypen in C.

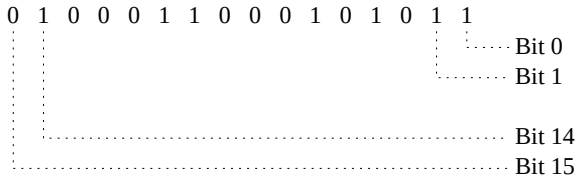
sind die Typen in dieser Tabelle in drei Gruppen unterteilt:

- Typen der ersten Gruppe sind standardisiert und haben immer die gleiche Größe, egal auf welchem Rechner/Betriebssystem das Programm übersetzt wurde oder läuft.
- Typen der zweiten Gruppe sind zwar auch standardisiert, ihre Größe hängt allerdings von dem verwendeten Compiler und dem darunterliegenden System ab. Im Rest dieser Vorlesung gehen wir davon aus, dass sowohl `int` als auch `unsigned int` 32 Bit groß sind (und damit den Typen `long` unf `unsigned long` entsprechen) und die Typen `size_t` und `ssize_t` den Typen `unsigned long` bzw. `long` entsprechen. Diese Annahme ist für alle 32-Bit-Architekturen (wie z.B. Pentium 1 bis 4) zutreffend. Den Typ `off_t` werden wir nicht benutzen.
- Die Variablen-Typen in der dritten Gruppe sind eine Erweiterung des `C`-Standards und gehören zur sogenannten ISOC99-Erweiterung. Sie werden dazu verwendet, um besonders große und besonders kleine Zahlenwerte darzustellen. Diese Variablentypen sind für uns in dieser Veranstaltung nicht von Bedeutung.

Außerdem ist beim Lesen der Tabelle folgendes zu beachten: Bei den Typen `float`, `double` und `long double` wurde als kleinster bzw. größter Wert jeweils die kleinste bzw. größte *normalisierte Zahl* angegeben. Das sind im Wesentlichen die kleinste und größte positive Zahl, die mit Variablen dieses Typs dargestellt werden können. Wer mehr über diese drei Typen erfahren will, der sei an die folgende Website verwiesen: *http://grouper.ieee.org/groups/754*.

Je nach Typ der Variable, wird der durch die Variable reservierte Speicherplatz anders interpretiert. Grundsätzlich gibt es dabei drei verschiedene Arten, wie die reservierten Bits interpretiert werden können:

Ganze Zahlen Bei dieser Interpretation werden die reservierten Bits als Zahlen im Dualsystem interpretiert. Da Bits genau einen der Zustände 0 oder 1 annehmen können (siehe Abschnitt 1.1) ist dies immer möglich. Dazu werden die Bits z.B. bei einer 16-Bit-Zahl wie folgt durchnumeriert:



Bei vorzeichenlosen Typen ergibt sich der Wert dann einfach als

$$2^{15} \cdot \text{bit15} + 2^{14} \cdot \text{bit14} + \ldots + 2^1 \cdot \text{bit1} + 2^0 \cdot \text{bit0}$$

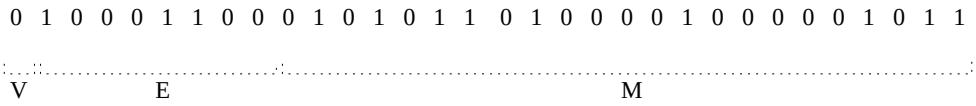
Negative Zahlen werden als sogenanntes ,,2er-Komplement" realisiert. Hierbei zeigt bit15 an, ob es sich um eine positive (bit15 = 0) oder eine negative Zahl (bit15 = 1) handelt. Die verbleibenden 15 Bits werden dann für die Codierung des Absolutbetrages der dargestellten Zahl verwendet. Details zu dieser Darstellung werden wir in der Übung behandeln. Analog erfolgt die Interpretation von ganzen Zahlen mit 8, 32 oder 64 Bits.

Die soeben beschriebene Interpretation betrifft Variablen der Typen `short`, `long`, `int`, `size_t`, `off_t`, `long long` sowie deren jeweilige vorzeichenlose Varianten.

Buchstaben Variablen des Typs `char` und `unsigned char` werden als Buchstaben (*engl.* ,,Characters", daher der Name des Typs) interpretiert. Zu diesem Zweck werden zunächst die dort gespeicherten Bits als ganze Zahl interpretiert und dann der entsprechende Buchstabe aus der sogenannten ASCII[1]-Tabelle ausgelesen (siehe Tabelle 2.3). In dieser Tabelle sind für die Zahlen 0 bis 127 jeweils Buchstaben/Kontrollzeichen festgelegt, die durch die jeweilige Zahl repräsentieren. Dabei ist insbesondere der Unterschied zwischen der Zahl 0 und dem Buchstaben '0' zu beachten!

Für die Zahlen von 128 bis 255 bzw. für die negativen Zahlen -128 bis -1 finden sich in dieser Tabelle keine Zeichen, weil der ASCII-Standard dafür ursprünglich keine definierte. Für diese Zeichen gibt es mehrere verschiedene standardisierte *Zeichensätze*, die den Zahlen jeweils ein Zeichen zuordnen. In Westeuropa wird normalerweise der ISO[2]-Zeichensatz 8859-1 verwendet (auch ,,ISO-latin1" genannt). Dieser Zeichensatz enthält die für die meisten westeuropäischen Sprachen benötigen Umlaute wie z.B. 'ä', 'ç' oder 'ô'. Wir werden in der Übung ein Programm schreiben, das eine entsprechende Tabelle ausgibt.

Gleitkomma Zahlen Bei dieser Interpretation werden die gespeicherten Bits in drei Gruppen interpretiert. Bei einer 32-Bit-`float`-Variablen wird, z.B. bit31 als Vorzeichen V ($V = 1 \iff$ bit31 $= 0$ und $V = -1 \iff$ bit31 $= 1$), die Bits 23 bis 30 als ganze Zahl E und die Bits 0 bis 22 ebenfalls als ganze Zahl M interpretiert:



Dabei wird E als Exponent und M als Mantisse bezeichnet. Die dargestellte Zahl berechnet sich dann als

$$V \cdot M \cdot 2^E.$$

In Wirklichkeit werden die Bitmuster in E und M nicht direkt als binäre Zahlen interpretiert, aber das Prinzip ist ähnlich.

Ähnlich erfolgt die Interpretation von 32-Bit-`double`- und 96-Bit-`long double`-Typen. Diese Art von Interpretation ist standardisiert und im IEEE[3]-Standard 754 niedergelegt. Zahlen die so interpretiert werden, werden deshalb auch als IEEE-754-Zahlen bezeichnet.

Übung: .....

1. Rechnen im Dualsystem,
2. 2er-Komplement,
3. Fließkomma-Zahlen,
4. Unterschied Fixpunkt vs. Gleitkomma,

[1] ASCII =**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange, z.B. *http://www.asciitable.com*
[2] ISO = International Organization for Standardization, *http://www.iso.org*
[3] IEEE = **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers, Inc., *http://www.ieee.org*

Code	Zeichen	Code	Zeichen	Code	Zeichen	Code	Zeichen
0	NUL	32	SPACE	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	99	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Hinweis für die Vorlesung:

Kontroll-Codes < 32 erklären: z.B. BEL = Glocke (Piepton), ACK = acknowlege, NAK = negative acknowledge (beides in Protokollen verwendet), CR = carriage return, LF = line feed, FF = form feed, HT = horizontal tab, VT = vertical tab (zur Ausgabe auf Drucker/Konsole), ESC = escape, DEL = delete (Backspace Taste)

Tabelle 2.3: Ausschnitt aus der ASCII-Tabelle.

5. 32-bit Muster interpretieren als
   (a) 4 `char` s,
   (b) 2 `short` s,
   (c) 2 `unsigned short` s
   (d) 1 `long`,
   (e) 1 `float`
6. ISO-latin1 Tabelle, außerdem `man iso_8859_1`

.....

2.3.2 Definition und Initialisierung

Wie wir bereits in dem Fibonacci-Beispielprogramm gesehen haben, können Variablen zu Beginn eines Blocks (siehe dazu auch Abschnitt 2.4 weiter unten) *definiert* werden. Dies geschieht durch die Angabe des Typs und der Liste der zu definierenden Variablen abgeschlossen durch ein Semikolon:

```
{
  int i1, i2, i3;
  char c1, c2, c3;
  float f1, f2, f3;

  ...
}
```

Dieser Code definiert drei Variablen vom Typ `int`, drei vom Typ `char` und drei vom Typ `float`. Die Definition entspricht der Reservierung einer entsprechenden Anzahl von Bits im Hauptspeicher. Nach der Definition enthalten die reservierten Bits zufällige (und damit nutzlose) Werte. Erst durch die *Initialisierung* der Variablen werden die Bits auf einen sinnvollen Wert gesetzt. Die Initialisierung geschieht durch die Zuweisung eines Wertes, wie z.B. in

```
i1 = 0;
c1 = 'a';
f1 = 3.14;
```

oder durch Zuweisung einer anderen Variablen:

```
i2 = i1;
c2 = c1;
f2 = f2;
```

Im zweiten Fall (Zuweisung einer anderen Variablen) sollte die zugewiesene Variable bereits initialisiert sein. Im ersten Fall (Zuweisung eines Wertes) werden auf der rechten Seite des Gleichheitszeichen sogenannte **Literale** (*engl.* „Literals") verwendet.

Die erlaubten Literale für ganze und Gleitkommazahlen sind dabei im Wesentlichen Zahlen im Wertebereichs des Typs der Variablen, an die zugewiesen wird. Bei Variablen des Typs `char` und `unsigned char` sind die erlaubten Literale alle Buchstaben aus Tabelle 2.3 oder der Zahlencode des entsprechenden Buchstabens. Die zugewiesenen Buchstaben müssen zu diesem Zweck allerdings in einfach Anführungszeichen eingeschlossen werden (vgl. Tabelle 2.3):

```
char b, c;
c = 'a'; /* same as c = 97 */
b = 65; /* same as b = 'A' */
```

Während sich die Zahlencodes allerdings noch leicht über die Tastatur eingeben lassen, ist das für manche Zeichen, für die es keine entsprechende Taste auf der Tastatur oder keine Darstellung auf dem Bildschirm gibt, schon etwas schwieriger. Da ist z.B. das Zeichen mit dem ASCII-Code 7. Wird dieses Zeichen gedruckt, dann piept der Computer – das Zeichen mit dem Code 7 ist der Piepton. Um dieses und einige andere Zeichen leichter eingeben zu können, wurden sogenannten Escape-Sequenzen eingeführt, die in Tabelle 2.4 aufgelistet sind. Außerdem muß der Backslash

Code	Kurzname	Name	Escape
0	NUL	Null	'\0'
7	BEL	Bell	'\a'
8	BS	Backspace	'\b'
9	HT	Horizontal Tab	'\t'
10	LF	Linefeed	'\n'
11	VT	Vertical Tab	'\v'
12	FF	Form Feed	'\f'
13	CR	Carriage Return	'\r'

Tabelle 2.4: Escape-Sequenzen für ASCII-Zeichen mit einem Code < 32.

('\') wegen seiner besonderen Rolle in den Escape-Sequenzen als '\\' angegeben werden.

Übung: .....

1. Unterschied: Definition vs. Initialisierung,
2. Fehlermeldung: „may be used uninitialised",
3. Zuweisung von zu großen/kleinen Zahlen, z.B. LONG_MAX and short, was passiert?

.....

2.3.3 Zuweisungen

Wie wir im vorigen Abschnitt bereits ansatzweise gesehen haben, kann man in `C` Variablen mit Hilfe des „="-Operators Werte zuweisen. Dabei kann man auf der rechten Seite nicht nur Literale sondern auch andere Variablen verweden, wie z.B. hier:

```
int i, j;
i = 1;
j = i;
```

Nach der Ausführung dieses Code-Fragments haben die beiden Variablen `i` und `j` beide den Wert 1. Bei Zuweisungen von einer Variablen an eine andere Variable sollten möglichst beide Variablen den gleichen Typ haben, ansonsten kann es leicht zu Problemen kommen. Betrachten wir z.B. das folgende Beispiel

```
long  u = 2147483647;
short s;

s = u;
```

Wie wir aus Tabelle 2.2 wissen, kann eine Variable des Typs `short` höchstens den Wert 32767 speichern. Der Wert von `s` ist nach der Zuweisung `s = u` also mit Sicherheit nicht 2147483647. Ähnliche Probleme entstehen, wenn man einer `float`-Variablen eine `double`-Variable zuweist oder vorzeichenlose mit vorzeichenbehafteten Typen mischt.

Übung: .....

1. Endlosschleife wie in cmp-signed-unsigned.c,
2. Zuweisung falscher (zu großer/kleiner) Werte

.....

2.3.4 Scopes

Jede Variable hat eine bestimmte Lebensdauer, den sogenannten **Scope** (*deut.* „Bereich"). Solange sich eine Variable in ihrem Scope befindet, kann sie benutzt werden. Innerhalb eines Scopes dürfen keine zwei Variablen mit dem gleichen Namen definiert sein.

Der Scope einer Variablen beginnt mir ihrer Definition und endet mit dem Block, zu dessen Beginn die Variable definiert wurde. Zur Verdeutlichung das folgende Beispiel:

```
{                           /* line 1 */
  int i;                    /* line 2 */
  i = 2;                    /* line 3 */
  {                         /* line 4 */
    int j, k;               /* line 5 */
    j = 3;                  /* line 6 */
    k = i + j;              /* line 7 */
  }                         /* line 8 */
  i = k;                    /* line 9 */
}                           /* line 10 */
```

Der Scope der Variablen `i` beginnt mir der Definition in Zeile 2 und endet mit der schließenden geschweiften Klammer in Zeile 10. Die Scopes der Variablen `j` und `k` beginnen mit der jeweiligen Definition in Zeile 5 und enden mit der schließenden geschweiften Klammer in Zeile 8. Die Anweisung `i = k` in Zeile 9 ist ein Fehler: der Scope von `k` endet bereits eine Zeile vorher, deshalb kann die Variable `k` in Zeile 9 nicht mehr verwendet werden.

Unglücklicherweise ist allerdings folgender Code erlaubt:

```
{                   /* line 1 */
  int i, j;         /* line 2 */
  i = 0;            /* line 3 */
  {                 /* line 4 */
    int i;          /* line 5 */
    i = 1;          /* line 6 */
  }                 /* line 7 */
  j = i;            /* line 8 */
}                   /* line 9 */
```

In Zeile 2 wird eine Variable `i` definiert, deren Scope bis Zeile 9 reicht. In Zeile 5 wird dann in einem neuen Scope nochmals eine Variable `i` definiert, deren Scope nun bis Zeile 7 reicht. Da in einem Scope auch alle Variablen von umschließenden Scopes ,,sichtbar" und benutzbar sind, existieren in Zeile 6 *zwei* Variablen namens `i`. Auf welche Variable bezieht sich nun die Anweisung `i = 1`? Die Antwort ist: auf die Variable, die in Zeile 5 definiert wurde. Die Regel in solchen Fällen ist:

> Wird in einem Scope eine Variable definiert, die bereits in einem äußeren umschließenden Scope definiert ist, dann ,,verdeckt" die Variable im inneren Scope, die aus dem äußeren Scope. D.h. wann immer im inneren Scope eine Variable mit diesem Namen referenziert wird, wird die Variable benutzt, die im inneren Scope definiert wurde. Dieses Verdecken von Variablen aus äußeren Scopes bezeichnet man als **Shadowing**.

Da sich die Zuweisung in Zeile 6 also auf die Variable bezieht, die in Zeile 5 definiert wurde, bleibt die in Zeile 2 definierte Variable `i` davon unberührt! In Zeile 8 gibt es dann wiederum nur eine Variable `i`, da der Scope der Variablen aus Zeile 5 schon beendet ist). In dieser Variablen ist nach wie vor der Wert 0 aus Zeile 3 gespeichert. Nach der Anweisung in Zeile 8 hat die Variable `j` also den Wert 0 (und nicht den Wert 1, wie man aufgrund von Zeile 6 vielleicht annehmen würde).

Obwohl solche Konflikte mit gleichen Variablennamen in einander umschließenden Scopes eigentlich eindeutig aufgelöst werden, ist es am besten solche Situationen von vorneherein zu vermeiden, da dadurch leicht schwer aufzufindende Fehler entstehen.

Übung: .....

1. Scoping,
2. Präzedenz bei gleichem Namen in enclosing scope und scope,
3. -Wshadow

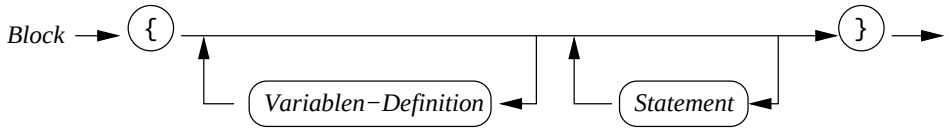
.....

2.4 Blöcke, Statements, Ausdrücke

Im vorherigen Abschnitt wurde bereits mehrfach der Begriff ,,Block" erwähnt. In diesem Abschnitt wollen wir nun etwas genauer erklären, wobei es sich darum nun genau handelt.

2.4.1 Blöcke

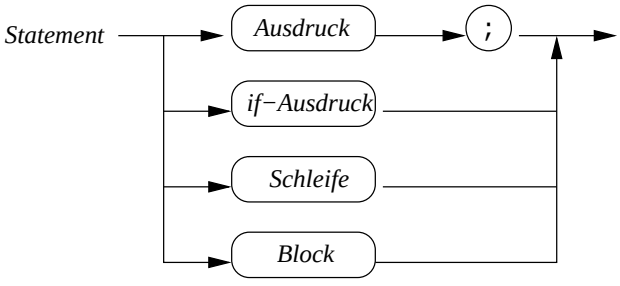
Vereinfacht ausgedrückt ist ein **Block** eine Liste von **Anweisungen** (*engl.* ,,Statements"), die ,,nach außen" wie eine einzige Anweisung wirkt. Blöcke werden im Normalfall immer dann verwendet, wenn die Syntaxregeln nur eine Anweisung erlauben, man aber mehrere Anweisungen angeben möchte. Der Aufbau eines Blocks läßt sich am einfachsten mit dem folgenden Schema beschreiben:



In diesem Schema ist ein Block irgendein Pfad von links nach rechts, in dem Kreise durchaus erlaubt sind. Demnach beginnt ein Block immer mit einer öffnenden geschweiften Klammer, gefolgt von einer beliebigen Anzahl (die auch Null sein darf) an Variablendefinitionen, gefolgt von einer (möglicherweise leeren) Liste von Statements, die schließlich durch eine schließende geschweifte Klammer abgeschlossen wird. Theoretisch ist also sogar ein leerer Block möglich, der nur aus einer öffnenden und einer schließenden Klammer besteht. In der Praxis ist das allerdings wenig sinnvoll.

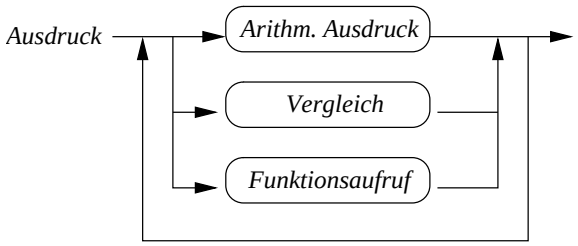
2.4.2 Statements und Ausdrücke

Im vorigen Abschnitt haben wir Statements als Teile von Blöcken kennengelernt. Auch Statements können wieder leicht durch ein Schema dargestellt werden, in dem ein Statement einem Weg von links nach rechts entspricht:



Demnach ist ein Statement entweder ein Ausdruck gefolgt von einem Semikolon, ein `if`-Ausdruck, eine Schleife oder ein Block. Der letzte Teil der Definition ermöglicht es uns also immer dann, wenn ein einzelnes Statement gefordert ist, eine Liste von Statements einzusetzen, wenn wir diese Liste in einen Block einschließen.

Blöcke kennen wir bereits aus dem vorherigen Abschnitt. `if`-Ausdrücke und Schleifen werden wir weiter unten behandeln. Bleiben also die Ausdrücke. Das ,,von links nach rechts"-Schema für einen Ausdruck sieht so aus:



Danach ist ein Ausdruck entweder eine Zuweisung wie z.B. `i = 3 * j` oder ein boolean-Ausdruck wie z.B. `i < j`. Die boolschen Operatoren, die in `C` unterstützt werden, können Tabelle 2.5 entnommen werden.

Operator		Bedeutung
<	binär	kleiner
<=	binär	kleiner oder gleich
>	binär	größer
>=	binär	größer oder gleich
==	binär	gleich
!=	binär	ungleich
!	unär	Verneinung

Tabelle 2.5: Boolsche Operatoren in `C`.

Werte von Ausdrücken

In `C` hat jeder Ausdruck einen Wert. Der Wert einer Zuweisung ist der zugewiesene Wert. D.h. der Wert des Ausdrucks `i = 3` ist 3. Dieser Wert kann nun in einem weiteren Ausdruck verwendet werden, wie z.B. in `j = (i = 3)` oder `j = 4 * (i = 3)`.

Dadurch ergibt sich die Frage, was bedeutet `i = (j < 3)` falls `i` eine Variable ist? Schließlich kann man keine Zahl mit einem Wert wie ,,wahr" oder ,,falsch" initialisieren. Die Lösung für dieses Problem ist in `C` (und in den meisten anderen Programmiersprachen auch) denkbar einfach. Der Wert eines boolschen Ausdrucks ist nicht etwa ,,wahr" oder ,,falsch" sondern eine Zahl. Wenn der boolsche Ausdruck wahr ist, dann ist der Wert des Ausdrucks 1, andernfalls ist der Wert des Ausdrucks 0. So kann der Wert jedes boolschen Ausdrucks immer als eine Zahl interpretiert werden. Umgekehrt kann in `C` auch jede Zahl als boolscher Ausdruck interpretiert werden: ist die Zahl ungleich 0, so wird die Zahl als wahrer Ausdruck, andernfalls als falscher Ausdruck gewertet.

Seiteneffekte von Ausdrücken

Neben seinem Wert hat ein Ausdruck in `C` meist auch einen sogenannten **Seiteneffekt** (*engl.* ,,Sideeffect"). So ist z.B. der Seiteneffekt des Ausdrucks `i = 3`, dass die Variable `i` den Wert 3 erhält. Im weiteren Verlauf der Veranstaltung werden wir sehen, dass Ausdrücke hauptsächlich wegen ihren Seiteneffekte und nicht so sehr wegen ihrer Werte verwendet werden.

Ein gutes Beispiel zur Unterscheidung von Wert und Seiteneffekt eines Ausdrucks sind die sogenannten ,,Pre-" und ,,Post-Inkrement"-Operatoren: Angenommen eine Variable `i` ist definiert,

dann sind sowohl `++i` als auch `i++` gültige Ausdrücke. Den ersten Ausdruck bezeichnet man Pre-, den zweiten als Post-Inkrement. Analoge Operatoren sind für die Dekrementierung definiert. Sowohl der Pre- als auch der Post-Inkrement-Operatoren erhöhen als Seiteneffekt den Wert der entsprechenden Variablen um eins. Allerdings ist der Wert des Pre-Inkrement-Ausdrucks der bereits erhöhte Wert der Variablen, während der Wert des Post-Inkrement-Ausdrucks der Original-Wert der Variablen ist. Tabelle 2.6 faßt diesen Sachverhalt noch einmal zusammen.

Ausdruck	Seiteneffekt	Wert
`i++`	`i` $\leftarrow 2$	1
`++i`	`i` $\leftarrow 2$	2
`i--`	`i` $\leftarrow 0$	1
`--i`	`i` $\leftarrow 0$	0

Tabelle 2.6: Wert und Seiteneffekte für Pre- und Post-Inkrement- und -Dekrement-Operatoren, für den Fall dass `i` $= 1$ gilt.

Übung: .....

1. Prä- und Postinkrement als spezielle Ausdrücke mit Seiteneffekten.

.....

Präzedenz von Operatoren

In `C` gelten für die Operatoren `+`, `-`, `*` und `/` die ganz normalen ,,Punkt-vor-Strich"-Regeln, die auch in der Mathematik gelten. Neben den vier eben genannten Operatoren gibt es in `C` aber noch eine ganze Reihe anderer Operatoren, für die jeweils Präzedenzen definiert sind. Obwohl die meisten Operatoren für uns zur Zeit noch unbekannt sind, sind in Tabelle 2.7 alle `C`-Operatoren mit ihren Präzedenzen aufgelistet. Stehen mehrere Operatoren der gleichen Präzedenz nebeneinander, so stellt sich die Frage ,,welcher Operator wird zuerst bearbeitet?" Diese Frage wird durch die sogenannte **Assoziativität** (*engl.* ,,associativity") beantwortet: ,,links nach rechts" bedeutet, dass bei Operatoren gleicher Präzedenz diese von links nach rechts abgearbeitet werden, ,,rechts nach links" bedeutet analog das Gegenteil.

Betrachten wir z.B. die zwei Ausdrücke `4 * 3 / 2` und `i = j = 1`. Nach den Regeln aus Tabelle 2.7 ist der erste Ausdruck äquivalent zu `(4 * 3) / 2` und der zweite äquivalent zu `i = (j = 1)`.

Wie in der Mathematik auch, haben runde Klammern die höchste Präzedenz, d.h. sie können dazu benutzt werden, vorgegebene Präzedenzen zu überschreiben, wie z.B. in `(3 + 1) * 5`.

2.5 Kontrollstrukturen

Ein wesentlicher Bestandteil eines jeden Computerprogramms ist die bedingte Ausführung. D.h. je nachdem welchen Wert gewisse Variablen haben und in welchem Zustand sich das System befindet werden verschiedene Operationen ausgeführt. Sehen wir uns z.B. den Befehl

Präzedenz	Operator	Assoziativität
hoch	'(', ')', '[', ']', '.', '→'	links nach rechts
	'++', '–', '+' (unär), '-' (unär), '!', ~, '(type)', '*pointer', '&variable', 'sizeof'	rechts nach links
↑	'*', '/', '%'	links nach rechts
	'+', '-'	links nach rechts
	'>>', '<<'	links nach rechts
	'<', '<=', '>', '>='	links nach rechts
⋮	'==', '!='	links nach rechts
⋮	'&'	links nach rechts
	'|'	links nach rechts
	'&&'	links nach rechts
	'||'	links nach rechts
↓	'?:'	rechts nach links
	'=', '+=', '-=', '*=', '/=', '%=', '&=', '~=', '|=', <<=, >>=	rechts nach links
niedrig	','	links nach rechts

Tabelle 2.7: Operator-Präzedenzen in `C`.

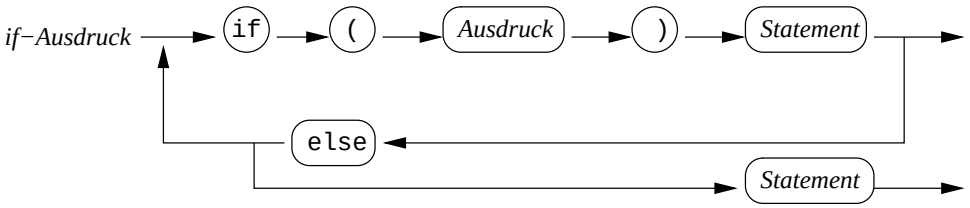
```
fb04305:  > rm data/Adressen.txt
```

an. Das Programm sucht nach einer Datei namens `data/Adressen.txt`. Falls diese Datei existiert, wird versucht sie zu löschen, andernfalls wird eine entsprechende Fehlermeldung ausgegeben. Beim Versuch, die Datei zu löschen, können wiederum zwei Fälle auftreten: die Datei kann nicht gelöscht werden (z.B. weil man nicht der Eigentümer der Datei ist) oder die Datei kann ohne Probleme entfernt werden. Im ersten Fall ist wieder eine entsprechende Fehlermeldung fällig.

Um Programme in dieser Art und Weise flexibel zu gestalten, sind Kontrollstrukturen notwendig, die dafür sorgen, dass bestimmte Programmteile nur dann ausgeführt werden, wenn entsprechende Bedingungen erfüllt sind. Wie solche Kontrollstrukturen in `C` aussehen und benutzt werden, werden wir in diesem Abschnitt erklären.

2.5.1 Alternativen

„Wenn ... dann ... andernfalls ..."-Alternativen wie wir sie soeben beschrieben haben, sind ein Hauptbestandteil der meisten Computerprogramme (und häufig rühren Fehler in Programmen daher, dass eine mögliche Alternative vom Programmierer nicht bedacht worden ist). In `C` können diese Alternativen mit sogenannten `if`-Anweisungen realisiert werden:



Ausdrücke werden dabei nach den in Abschnitt 2.4.2 beschriebenen Regeln in ,,wahr" oder ,,falsch" übersetzt (sofern es sich nicht schon um boolsche Ausdrücke handelt). Im Programm könnte eine solche `if`-Anweisung z.B. wie folgt aussehen:

```
if (i < 3) printf("i ist kleiner als 3!");
else if (i > 3) printf("i ist größer als 3!");
else printf ("i ist genau 3!");
```

Falls der Wert der Variablen `i` während der Ausführung dieses Code-Fragments kleiner als 3 ist, so wird ,,i ist kleiner als 3!" ausgegeben. Ist dies nicht der Fall und `i` ist größer als 3, so wird ,,i ist größer als 3!" ausgegeben. Trifft keiner der beiden Fälle zu, so wird ,,i ist genau drei!" ausgegeben.

Selbstverständlich kann man auch mehrere Anweisungen in einer Verzweigung benutzen. Dazu muss man sich dann eines Blocks bedienen.

Übung: .....

1. `switch`?

.....

2.5.2 Schleifen

Neben den oben beschriebenen Alternativen tauchen in den meisten Computerprogrammen auch bestimmte Arten von Wiederholungen auf. Diese Wiederholungen lassen sich meist in der Form ,,solange ... wahr ist, tue ..." wiedergeben. Diese Sprachkonstrukte werden in `C` mit Hilfe von sogenannten **Schleifen** (*engl.* ,,Loops") nachgebildet. Die einfachste dieser Schleifen ist die `while`-Schleife:

Solange *Ausdruck* einen wahren Wert hat, wird *Statement* wieder und wieder ausgeführt. Vor jeder Ausführung von *Statement* wird erneut geprüft, ob *Ausdruck* wahr ist und die Wiederholung des Statements wird gegebenenfalls abgebrochen.

Auch hier gilt wieder: eine Liste von Statements kann ausgeführt werden, indem man einen Block verwendet. Als Code-Beispiel für die `while`-Schleife mag das Fibonacci-Programm vom Beginn dieses Kapitels dienen.

Eine Variante der `while`-Schleife ist die `do`-Schleife:

Hier wird zunächst *ein*mal *Statement* ausgeführt, bevor *Ausdruck* überprüft wird. Ist *Ausdruck* wahr, so wird *Statement* ein weiteres Mal ausgeführt und *Ausdruck* nochmals überprüft usw. Sobald *Ausdruck* bei der Überprüfung keinen wahren Wert mehr hat, wird die Bearbeitung der Schleife abgebrochen,

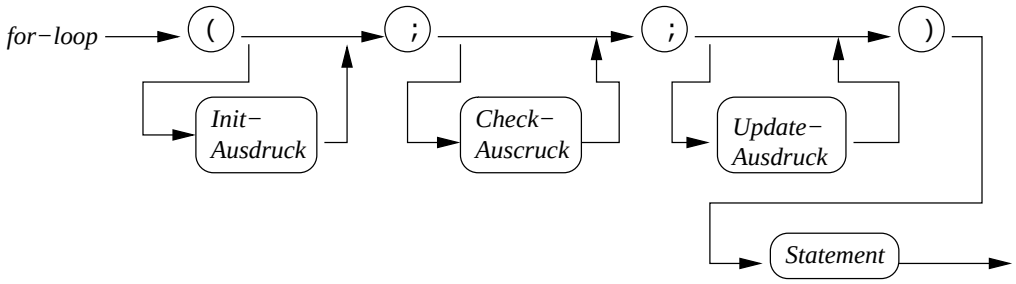
Für den speziellen Fall, daß ein Statement (oder Block) eine bestimmte Anzahl oft wiederholt werden soll, kann die `for`-Schleife benutzt werden:

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

Dieses Beispiel würde auf der Konsole den Text

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

ausgeben. Das formale Schema für eine `for`-Schleife sieht wie folgt aus:



Zu Beginn der Schleife wird *Init-Ausdruck* ausgeführt. Danach wird getestet, ob *Check-Ausdruck* wahr ist. Ist dem so, so wird *Statement* ausgeführt, andernfalls wird die Schleife direkt abgebrochen. Nach jeder Ausführung von *Statement* wird zunächst *Update-Ausdruck* ausgeführt und dann erneut *Check-Ausdruck* getestet. Ist dieser nach wie vor wahr, dann wird *Statement* ein weiteres Mal ausgeführt, andernfalls wird die Bearbeitung der Schleife abgebrochen.

Im Gegensatz zur `while`-Schleife darf bei der `for`-Schleife jeder der drei Ausdrücke auch der *leere* Ausdruck sein. Dabei wird ein leerer Check-Ausdruck immer als wahr gewertet und bei leerem Init- bzw. Update-Ausdruck wird einfach nichts getan. Demnach ist

```
for (;;) {}
```

also eine gültige `for`-Schleife, und zwar eine, die für immer läuft!

Übung: .....

1. `break`

2. `continue`

.....

2.5.3 Verknüpfung boolscher Ausdrücke

In Abschnitt 2.4.2 hatten wir bereits gesehen, was boolsche Ausdrücke sind. Wir erinnern noch einmal daran, dass jede Zahl als boolscher Ausdruck verwendet werden kann und umgekehrt. Dabei gilt für eine Zahl n, die als ein solcher Ausdruck A verwendet wird

$$\begin{aligned} A \text{ ist wahr} &\iff n \neq 0 \\ A \text{ ist falsch} &\iff n = 0. \end{aligned}$$

Häufig möchte man bei der Verwendung von Kontrollstrukturen allerdings nicht nur eine Bedingung prüfen, sondern mehrere. Zu diesem Zweck gibt es in `C` Operatoren, mit deren Hilfe man logische Bedingungen verknüpfen kann. Diese Operatoren sind

- `&&` logisches ,,und"
- `||` logisches ,,oder"
- `!` Verneinung

Nehmen wir z.B. an, dass wir drei Variablen `a`, `b` und `c` vom Typ `int` haben und betrachten uns den folgenden Ausdruck:

```
a && (!b || c > 3).
```

Dieser Ausdruck ist genau dann wahr, wenn `a` ungleich 0 ist und entweder `b` gleich 0 oder `c` grösser als 3 ist oder beides zutrifft. Entsprechend ist der Ausdruck falsch, falls `a` gleich 0 ist oder `b` ungleich 0 und `c` kleiner oder gleich 3 ist.

Betrachten wir noch einmal den Ausdruck von oben, dann stellen wir fest, dass er nie wahr sein kann, falls `a` gleich 0 ist. In diesem Fall müßten wir den zweiten Teil des Ausdrucks also gar nicht auswerten. Ebenso ist z.B. der Ausdruck

```
(a && b) || c >= 4
```

schon dann wahr, wenn `a` und `b` beide ungleich 0 sind. In diesem Fall, können wir uns den Test `c >= 4` sparen.

Tatsächlich wird in `C` genau nach diesem Prinzip vorgegangen: Sobald klar ist, dass ein logischer Ausdruck seinen Wert nicht mehr ändert, wird die Bearbeitung des Ausdrucks abgebrochen. Man nennt diese Art von Auswertung **short-circuit evaluation**. Beim Verknüpfen von boolschen Ausdrücken muss man sich dieser Auswertungsstrategie immer bewußt sein, wie das folgende Beispiel zeigt:

```
a && b++ == 3.                                                    (*)
```

Ist `a` gleich 0, dann ist der Ausdruck immer falsch und `b++ == 3` wird nicht ausgewertet. Ist aber `a` ungleich 0, dann hängt der Wert des Ausdrucks von `b++ == 3` ab. Bei der Auswertung von `b++ == 3` wird aber nicht nur geprüft, ob `b` den Wert 3 hat, sondern als Seiteneffekt auch noch `b` um 1 erhöht!

Aufgrund der short-circuit Auswertung wird `b` also nur dann erhöht, wenn beim Auswerten von $(*)$ die Variable `a` einen Wert ungleich 0 hat!

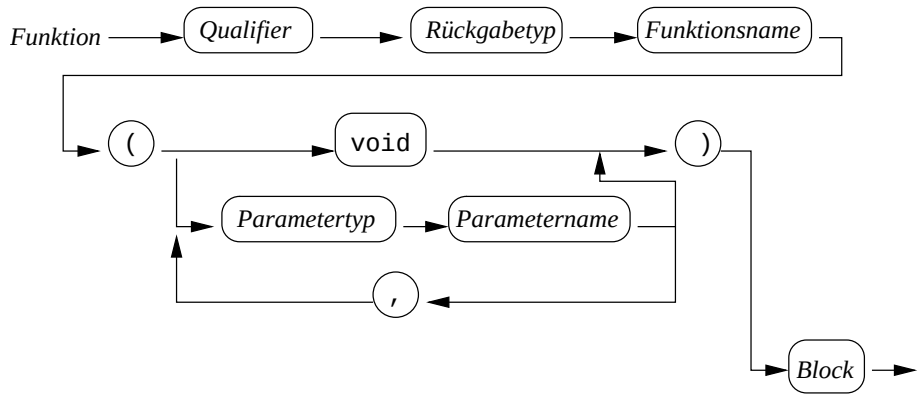
Das Prinzip der short-circuit Auswertung wird uns später bei der Behandlung von Zeigern noch häufiger begegnen.

2.6 Funktionen

In den meisten Computerprogrammen gibt es Befehlssequenzen, die immer wieder gebraucht werden. Wir haben als Beispiel bereits die Funktion `printf` kennengelernt. Diese Funktion sorgt dafür, dass der übergebene Text auf dem Bildschirm ausgegeben wird. Damit man nicht jedesmal, wenn man etwas auf dem Bildschirm ausgeben will, alle dafür notwendigen Befehle einzeln eintippen muß, wurden diese in der Funktion `printf` zusammengefaßt. Anstelle der einzelnen Befehle können wir deswegen einfach immer diese Funktion zusammen mit den entsprechenden Argumenten aufrufen.

Eine Funktion kann man sich vorstellen als einen Block, dem ein Name gegeben wurde: Der Block enthält die Liste der Anweisungen, die durch die Funktion zusammengefaßt werden und der Name erlaubt es dem Programmierer diesen Block aufzurufen. Außerdem kann eine Funktion mit Hilfe des Schlüsselwortes `return` einen Wert zurückgeben. Dieser Wert kann z.B. das Ergebnis einer Berechnung sein oder auch anzeigen, dass es einen Fehler gab.

Syntaktisch folgt die Definition einer Funktion dem folgenden Schema:



wobei wir in dieser Veranstaltung die *Qualifier* nicht weiter berücksichtigen: diese Komponente wird bei uns immer leer sein.

Ein Beispiel für Funktionen haben wir bereits in den beiden Beispielprogrammen zu Beginn dieses Kapitels gesehen: die Funktion `main`.

Die Parameterliste einer Funktion dient im Grunde dazu, Variablen für die Funktion zu definieren, die beim Aufruf der Funktion automatisch initialisert werden. Nun gibt es aber auch Funktionen, die entweder keine Parameter benötigen oder die keinen Wert zurückgeben. Um dies anzuzeigen, benutzt man den Datentyp `void`. Im strengen Sinn, ist `void` eigentlich kein Datentyp, denn man kann keine Variablen vom Typ `void` definieren. Bei der Definition einer Funktion kann dieses Schlüsselwort aber anstelle der im Schema geforderten Typen stehen:

- Die Funktion kann `void` zurückgeben. Der Wert eines entsprechenden Funktionsaufrufes ist dann ebenfalls `void` und kann weder einer Variablen zugewiesen werden, noch als ein boolscher Wert interpretiert werden.

- Die Parameterliste einer Funktion kann `void` sein. Eine solche Parameterliste ist dann eben einfach leer. Der Einfachheit halber kann man statt `(void)` für die Parameterliste auch `()` schreiben. Wir raten der Klarheit halber allerdings zur ersten Variante.

Nachdem wir Funktionen nun aus syntaktischer Sicht beleuchtet haben, wollen wir uns nun noch klar machen, was Funktionen für die Programmausführung bedeuten. Ähnlich wie Alternativen und Schleifen weichen Funktionsaufrufe von der seriellen Abarbeitung der Anweisungen ab. Während Alternativen und Schleifen aber zu einer anderen Zeile/einer anderen Anweisung springen, um dort die Ausführung des Programms fortzusetzen, verzweigen Funktionen an eine andere Stelle des Codes, um dort einige Befehle auszuführen und dann wieder an die Stelle der Verzweigung zurückzukehren. Wir haben versucht, diese Unterschiede in Abbildung 2.1 zu verdeutlichen:

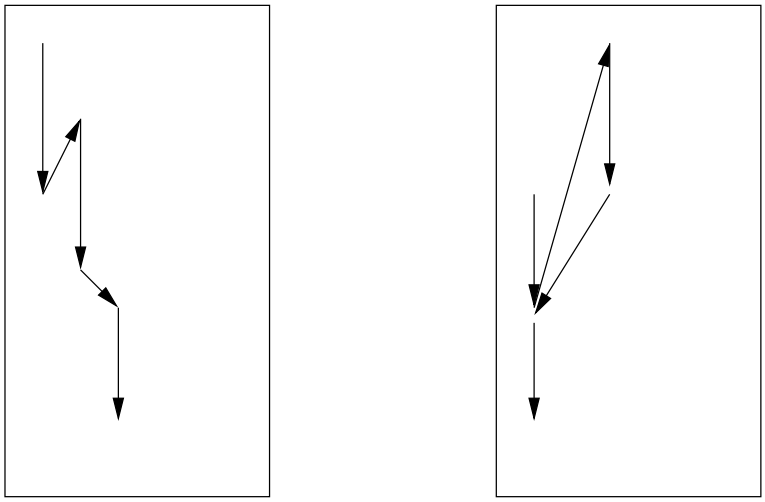


Abbildung 2.1: Ausführung von Alternativen/Schleifen (links) und Funktionsaufrufen (rechts).

- Bei Alternativen und Schleifen wird wie gesagt die serielle Abarbeitung der Befehle unterbrochen, das Programm springt an eine andere Stelle des Codes und setzt dort die serielle Bearbeitung der Befehle fort. Insbesondere merkt sich das Programm weder von wo der Sprung ausgegangen ist, noch dass überhaupt ein Sprung stattgefunden hat.
- Bei Funktionsaufrufen springt das Programm zunächst an die Stelle, an der die entsprechende Funktion definiert ist und führt dort die in der Funktion zusammengefaßten Anweisungen aus. Danach springt das Programm wieder *zurück* zu der Stelle, von der aus der Funktionsaufruf erfolgte und setzt die Bearbeitung mit der nächsten Anweisung fort. In diesem Fall *merkt* sich das Programm also, dass ein Sprung stattgefunden hat und von wo dieser ausgegangen ist.

  Das rechte Bild in Abbildung 2.1 läßt sich übrigens noch beliebig verkomplizieren, wenn man berücksichtigt, dass Funktionen ihrerseits wieder andere Funktionen oder sich selber aufrufen können.

Wie bereits erwähnt, dienen die Parameter einer Funktion dazu, eine Liste von Variablen zur Verfügung zu stellen, die vom aufrufenden Programmteil bereits initialisiert wurden. Diese Variablen werden im Normalfall dazu benutzt, die Funktion zu steuern bzw. ihr mitzuteilen, was genau sie nun tun soll.

Betrachten wir uns dazu ein kurzes Beispiel: die Funktion

```
int square (int x)
{
  int result = x * x;
  return result;
}
```

berechnet allgemein das Quadrat einer Zahl. Rufen wir diese Funktion mittels

```
int a;
a = square(5);
```

auf, dann wird der Parameter `x` von `square` beim Aufruf mit `5` initialisiert. Demnach ,,weiss" die Funktion, dass sie 5^2 berechnen soll. Das Ergebnis dieser Berechnung wird in `result` zwischengespeichert und dann mittels `return` zurückgegeben. Der von der Funktion zurückgegebene Wert (also 25) wird schließlich der Variablen `a` zugewiesen.

Beim Verwenden der `return`-Anweisung sind folgende Dinge zu beachten:

1. Der zurückgegebene Wert muss den Typ haben, der als Rückgabetyp angegeben wurde. Bei `void`-Funktionen kann man einfach ,,`return;`" schreiben (d.h. ohne einen Wert).

2. Die `return`-Anweisung muss nicht notwendigerweise die letzte Anweisung in der Funktion sein. Häufig werden z.B. Konstrukte der Form

   ```
   if ( /* condition 1 */ ) {
     return ...;
   }
   if ( /* condition 2 */ ) {
     return ...;
   }
   return ...;
   ```

   verwendet.

3. Ist die letzte Zeile der Funktion keine `return`-Anweisung, so wird so getan, als ob nach der letzten Zeile noch `return;` stünde. Für `void`-Funktionen ist das kein Problem. Bei Funktionen, die einen Wert zurückgeben ist dies allerdings ein Fehler.

Übung: .....

1. Deklaration vs. Definition,

.....

2.7 Benutzerdefinierte Typen

In Abschnitt 2.3 haben wir bereits die Möglichkeit kennengelernt, mit Variablen Speicher zu reservieren und den reservierten Bereich auf eine bestimmte Weise zu interpretieren. So konnten wir leicht Zahlen und/oder Buchstaben (zwischen)speichern. In den meisten Anwendungen reicht es aber nicht aus, einfach nur einzelne Zahlen oder Buchstaben zu speichern. Vielmehr möchte man mehrere Zahlen, die aufgrund bestimmter Eigenschaften zusammengehören, auf einmal speichern und bearbeiten.

Nehmen wir z.B. einmal an, dass wir mit komplexen Zahlen arbeiten wollen. Standardmäßig gibt es in `C` keinen Datentyp, der zum Speichern und Verarbeiten von komplexen Zahlen vorgesehen ist. Da wir Real- und Imaginärteil der Zahl behandeln müssen, bräuchten wir also zwei `double`-Variablen, um eine komplexe Zahl darzustellen. Z.B. so:

```
{
  double x_real;
  double x_imag;

  x_real = 1.0;
  x_imag = 2.0;
}
```

Wie man sich leicht vorstellen kann, ist diese Art sehr umständlich und fehleranfällig. Leichter und besser kann man das Ziel, eine komplexe Zahl zu bearbeiten, mit einer sogenannten `struct` erreichen:

```
{
  struct complex {
    double real;
    double imag;
  };

  struct complex x = { real:1.0, imag:2.0 };
}
```

Dieses Codefragment tut im Wesentlichen genau das gleiche, wie das vorhergehende Fragment, allerdings auf einem wesentlich eleganteren Weg:

- Zunächst wird eine `struct` namens `complex` definiert. Diese `struct` hat zwei **Felder** (*engl.* „Fields" oder *engl.* „Members"), `real` und `imag`, die beide vom Typ `double` sind.
- Ab der Definition dieser `struct` gibt es nun zusätzlich zu den bereits bekannten Datentypen wie `int`, `char` usw. einen neuen Datentyp namens `struct complex` (das Schlüsselwort `struct` gehört zum Datentyp dazu). Dieser kann nun genauso wie die eingebauten Datentypen verwendet werden.
- Nachdem wir `struct complex` definiert haben, definieren wir im Beispiel eine Variable von diesem Typ und initialisieren sie gleich: `struct complex x = {real:1.0, imag:2.0}`.

Der Literal, den wir zur Initialisierung benutzen ist einfach eine in geschweifte Klammern eingeschlossene Liste von zuzuweisenden Werten. Der Einfachheit halber ist auch `struct complex x =` $\{1.0, 2.0\}$ als Initialisierung erlaubt. Dabei werden die angegebenen Werte in der gegebenen Reihenfolge den Feldern der Struktur zugeordnet.

- Um nun auf die einzelnen Felder `real` und `imag` der Variablen `x` zugreifen zu können, benutzen wir den Operator „.": `x.real` bzw, `x.imag`, liefert die Felder `real` bzw. `imag` der Variablen `x`. Wir können demnach z.B. mit `x.real = 3.0` den Realteil auf 3 setzen.

Die Felder in einer `struct` können beliebige Datentypen haben, insbesondere also auch selbst wieder eine `struct` sein. So lassen sich beliebig komplizierte Datentypen konstruieren.

Übung: .....

1. Definiere `struct complex`
2. Schreibe Funktionen, die komplexe Zahlen addieren, multiplizieren, dividieren usw.

.....

2.8 Zeiger, Arrays und Strings

Bisher haben wir nur Variablen kennengelernt, die zum Reservieren und Beschreiben/Auslesen von Speicherzellen dienten. In diesem Abschnitt wollen wir nun eine weitere wichtige Klasse von Variablen vorstellen, sogenannte **Zeiger** (*engl.* „Pointer").

2.8.1 Zeiger

Wie wir bereits in Abschnitt 1.1.2 erläutert haben, besteht der Hauptspeicher im Computer aus einer Vielzahl von Bits, die wiederum in Gruppen zu je acht – den sogenannten Bytes – zusammengefaßt werden. Jedem Byte im Hauptspeicher ist zusätzlich eine „Adresse" zugeordnet: das erste Byte hat die Adresse 0, das zweite die Adresse 1 usw.

Zeiger dienen nun nicht dazu, Zahlen oder Buchstaben zu speichern, sondern zum Speichern von Adressen. Dieses Prinzip erklären wir wieder an einem kurzen Code-Beispiel:

```
int i;                                      /* line 1 */
int *i_ptr;                                 /* line 2 */
                                            /* line 3 */
i = 4;                                      /* line 4 */
i_ptr = &i;                                 /* line 5 */
printf("%d, %d, %p\n", i, *i_ptr, i_ptr);   /* line 6 */
                                            /* line 7 */
*i_ptr = 5;                                 /* line 8 */
printf("%d, %d, %p\n", i, *i_ptr, i_ptr);   /* line 9 */
```

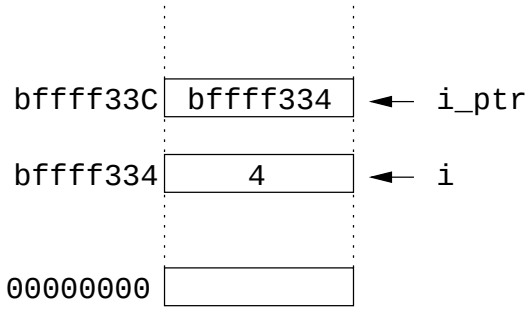
In diesem Code-Fragment passiert Folgendes

Zeile 1 Hier wird auf die bereits bekannte Art und Weise eine Variable vom Typ `int` definiert.

Zeile 2 Diese Zeile definiert den Zeiger `i_ptr`. Diese Variable hat den Typ `int*` (Whitespaces zwischen `int` und `*` werden ignoriert) und dient dazu, die Adresse einer `int`-Variablen zu speichern.

Zeile 4 Dieses Statement weist in altbekannter Weise der Variablen `i` den Wert 4 zu.

Zeile 5 Hier wird nun die Zeigervariable `i_ptr` initialisiert: Der Ausdruck `&i` liefert die Adresse der Bytes im Hauptspeicher zurück, die für die Variable `i` reserviert wurden. Diese Adresse wird der Variablen `i_ptr` zugewiesen. Wir sagen in diesem Fall dann häufig „`i_ptr` zeigt auf `i`". Angenommen wir stellen Speicheradressen als hexadezimale Zahlen dar, dann haben wir nach dieser Zeile folgendes Bild des Hauptspeichers:



Die Variable `i` ist an Adresse `bffff334` im Hauptspeicher abgelegt, die Variable `i_ptr` an Adresse `bffff33C` (beim Nachprogrammieren dieses Beispiels können andere Adressen auftreten, das Prinzip bleibt aber das gleiche).

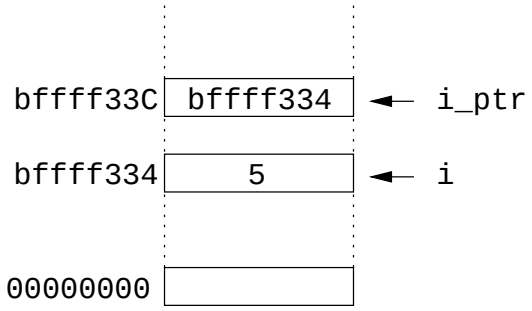
Wie gezeigt speichern die Bytes an Adresse `bffff334` (d.h. die Variable `i`) den Wert `4`. Dagegen speichert die Variable `i_ptr` den Wert `bffff334`, das ist nämlich die Adresse von `i`.

Zeile 6 Diese Statement sorgt dafür, dass

```
4, 4, 0xbffff334
```

auf der Konsole ausgegeben wird. Dabei gibt es im Vergleich zu den `printf`-Anweisungen, die wir bereits kennengelernt haben, zwei wesentliche Neuheiten: das Format `%p` und das Konstrukt `*i_ptr`. Das Format `%p` dient dazu, Zeigervariablen auszugeben. Es sorgt dafür, dass die in der entsprechenden Zeigervariablen gespeicherte Adresse in hexadezimaler Form (mit vorgestelltem `0x`) ausgegeben wird. Das Konstrukt `*i_ptr` liefert den Wert der Variablen, auf die `i_ptr` zeigt. In unserem Fall zeigt `i_ptr` ja auf die Variable `i`, die wiederum auf 4 gesetzt ist. Demnach liefert `*i_ptr` also 4.

Zeile 8 Ähnlich wie in Zeile 6 wird hier der Ausdruck `*i_ptr` verwendet. Diesmal allerdings nicht um den Wert der Variablen, auf die `i_ptr` zeigt, zu lesen, sondern um diesen Wert zu setzen. Die Variable `i` wird also durch dieses Statement auf 5 gesetzt:



Die Variable `i_ptr` selbst bleibt durch diese Operation unverändert.

Zeile 9 Diese Zeile funktioniert exakt wie Zeile 6, gibt diesmal aber aufgrund der vorher gemachten Änderung der Variablen `i` den Text

```
5, 5, 0xbffff334
```

aus.

2.8.2 Zuweisen von Zeigern

In Abschnitt 2.3.3 haben wir bereits kurz das Zuweisen von Variablen an Variablen anderen Typs angesprochen. Während jedoch das Zuweisen von `float` an `int` durchaus erlaubt ist, ist das Zuweisen von `float*` an `int*` nicht erlaubt. Der Compiler übersetzt solchen Code zwar, gibt aber eine Warnung des Typs

```
warning: assignment from incompatible pointer type
```

aus. Obwohl es einige wenige Spezialfälle gibt, in denen es sinnvoll ist, Zeiger unterschiedlichen Typs einander zuzuweisen, werden wir solche Zuweisungen in dieser Veranstaltung als ,,verboten" ansehen.

2.8.3 Arrays

Wie man einzelne Variablen anlegt, haben wir bereits zu Genüge betrachtet. Häufig möchte man aber nicht nur eine einzige Variable, sondern mehrere Variablen, die logisch zusammenhängen. Wollen wir z.B. mit Vektoren im $\mathbb{R}^3$ arbeiten, so brauchen wir jeweils für die x-, y- und z-Koordinaten der Vektoren eine `int` Variable. Wir kennen bereits zwei Wege, um dies zu bewerkstelligen:

1. Wir repräsentieren einen Vektor `i` nicht als eine Variable, sondern als drei, nämlich `int i_x, i_y, i_z`.
2. Wir definieren eine `struct` mit den ensprechenden Feldern

```
struct vec_3 {
  int x;
  int y;
  int z;
};
```

und definieren dann einen Vektor `i` mittels `struct vec_3 i`.

Eine dritte Möglichkeit ergibt sich durch die Benutzung eines sogenannten **Arrays**: Im Gegensatz zu einer Variablen ist ein Array eine bestimmte Anzahl von Variablen vom gleichen Typ. In unserem Fall würden wir einen Vektor `i` wie folgt definieren

```
int i[3];
```

Dies legt fest, dass `i` ein Array von 3 Variablen des Typs `int` ist. `i` reserviert im Hauptspeicher nicht nur Platz für eine, sondern gleich für 3 `int`-Variablen. Die erste dieser `int`-Variablen spricht man mit `i[0]` an, die zweite mit `i[1]` usw. Implementieren wir dreidimensionale Vektoren mit Hilfe eines Arrays, dann läßt sich z.B. das Skalarprodukt sehr einfach ausrechnen:

```
int a[3] = { 1, 2, 3 };
int b[3] = { 4, 5, 6 };
int i, prod;

prod = 0;
for (i = 0; i < 3; i++) prod += a[i] * b[i];
```

Hätten wir dagegen eine `struct` zur Implementierung eines Vektors verwendet, so hätte diese Berechnung wesentlich komplizierter ausgesehen. Insbesondere läßt sich das obige Beispiel leicht auf andere Dimensionen k anpassen (man muss nur jeweils die 3 durch k ersetzen). Beide Beispiele zeigen außerdem, dass die Initialisierung eines Arrays analog der Initialisierung einer `struct` verläuft.

Wie bereits erwähnt, reserviert `int i[3]` Platz für drei Variablen vom Typ `int` hintereinander. Somit reicht es aus, die Adresse der ersten dieser Variablen und die Länge des Arrays zu kennen. Aus diesem Grund, behandelt `C` einen Array von `int` s und einen Zeiger auf eine `int`-Variable gleich:

```
int A[3] = { 1, 2, 3 };
int *B;

printf("%d\n", A[0]);   /* prints A[0] (i.e. '1') */

B = A;
printf("%d\n", B[0]);   /* prints A[0] (i.e. '1') */
printf("%d\n", *B);     /* prints A[0] (i.e. '1') */

*B += 10;               /* increments A[0] */
printf("%d\n", A[0]);   /* prints A[0] (i.e. '11'); */
```

Mit diesem Wissen können wir nun eine Funktion implementieren, die allgemein das Skalarprodukt zweier Vektoren ausrechnet:

```
int scalar_product (int* a, int* b, int dim)
{
  int i, prod;

  prod = 0;
  for (i = 0; i < dim; i++) prod += a[i] * b[i];

  return prod;
}
```

Die Funktion `scalar_product` bekommt jeweils einen Zeiger auf das erste Element des ersten und zweiten Vektors (Arrays) sowie die Dimension der Vektoren übergeben. Wie bereits im oberen Codebeispiel gesehen wird dann das Skalarprodukt Komponente für Komponente ausgerechnet und in `prod` akkumuliert. In einer Anwendung könnte die Funktion dann z.B. so verwendet werden:

```
int A3[3] = { 1, 2, 3 }, B3[3] = { 4, 5, 6 };
int A4[4] = { 1, 2, 3, 4 }, B4[4] = { 5, 6, 7, 8 };
int a = 1, b = 2;
int p1, p2, p3;

p1 = scalar_product(A3, B3, 3);
p2 = scalar_product(A4, B4, 4);
p3 = scalar_product(&a, &b, 1);
```

Interessant ist dabei besonders die Berechnung von `p3`: hier wurde wieder ausgenutzt, dass ein Array von `C` wie ein Zeiger auf das erste Element behandelt wird (und umgekehrt). Wir können demnach einen Zeiger auf eine Variable als einelementiges Array des entsprechenden Typs betrachten.

Übung: .....

1. Implementiere Vektor in $\mathbb{R}^4$ als `struct` und als `Array` und einige Funktionen. Vergleiche.
2. Zuweisung von Arrays.
3. Arrays von Arrays.

.....

2.8.4 Dynamisches Allokieren von Speicher

Bisher haben wir Speicherplatz immer dadurch reserviert, dass wir zu Beginn eines Blocks die entsprechenden Variablen definiert haben. Dies ist aber nicht immer möglich. Nehmen wir z.B. einmal an, wir wollen eine Funktion

```
int* add_to_array (int* array, int len, int add);
```

implementieren. Diese Funktion soll zu jedem Element eines gegebenen Arrays von `int`-Variablen einen festen Wert addieren und das Ergebnis in einem neuen Array zurückliefern (d.h. das Eingabearray soll nicht verändert werden). Zu diesem Zweck wird der Funktion das Eingabearray (`array`), die Länge dieses Arrays (`len`) sowie das zu addierende Element (`add`) übergeben. Zum Zeitpunkt, zu dem das Programm übersetzt wird, ist also nicht bekannt, wieviele Elemente das Eingabearray (und damit auch das Ausgabearray) haben wird. Die erste Idee, dieses Problem zu lösen ist wie folgt:

```
int* add_to_array (int* array, int len, int add)
{
  int i;
  int ret[len];

  for (i = 0; i < len; i++) ret[i] = array[i] + add;

  return ret;
}
```

D.h. wir benutzen keine Zahl als Arraylänge, sondern den Wert der Variablen `len`. Das Problem an diesem Ansatz ist der Scope von `ret`: Mit dem Ende der Funktion `add_to_array` endet auch der Scope von `ret`, d.h. der Speicher, der für diese Variable reserviert war, wird am Ende von `add_to_array` wieder freigegeben und andere Daten können auf diesem Bereich gespeichert werden. Die Funktion gibt also einen Zeiger auf einen nicht (länger) reservierten und damit uninitialisierten Speicherbereich zurück. Damit ist der Rückgabewert dieser Funktion dem Zufall unterworfen.

Um das Problem korrekt zu lösen, bedient man sich der Funktion `malloc`, die in der Headerdatei `stdlib.h` deklariert ist:

```
#include <stdlib.h>

int* add_to_array (int* array, int len, int add)
{
  int i;
  int* ret;

  ret = malloc(sizeof(int) * len);

  for (i = 0; i < len; i++) ret[i] = array[i] + add;

  return ret;
}
```

In diesem Codebeispiel reserviert die Funktion `malloc` Platz für `len` Variablen vom Typ `int` hintereinander, d.h. ein `int`-Array der Länge `len`. Der Unterschied zum vorherigen Ansatz ist der Folgende: Mit dem Ende des Scopes von `ret` wird der durch `malloc` reserviert Speicherbereich *nicht* wieder freigegeben. Demnach zeigt die von `add_to_array` zurückgegebene Adresse nach wie vor auf einen reservierten Speicherbereich.

Wieso das? Ganz einfach: im zweiten Code-Fragment ist `ret` gar nicht als Array, sondern lediglich als Zeiger auf eine `int`-Variable definiert. Wie wir aus Abschnitt 2.8.3 wissen, unterscheidet `C` im Wesentlichen nicht zwischen einem Array und einem Zeiger auf das erste Element des Arrays. Für `ret` wird also insbesondere kein Array reserviert, sondern lediglich der Platz, der nötig ist, um eine Adresse im Hauptspeicher aufnehmen zu können. Mit `ret = malloc(...)` wird `ret` dann die Startadresse eines *dynamisch* reservierten Speicherbereichs zugewiesen. Von diesem Punkt an, kann `ret` wie ein Array der Länge `len` benutzt werden. Am Ende von `add_to_array` ist dann lediglich bekannt, dass in `ret` eine Adresse steht, nicht allerdings woher diese kam (`ret` hätte z.B. auch mit `ret = &i` initialisiert worden sein, obwohl das hier keinen Sinn gemacht hätte). Aus diesem Grunde wird auch nur der Speicher freigegeben, der für `ret` reserviert war und nicht der, der durch `malloc` angefordert wurde.

Da der Hauptspeicher explizit angefordert worden ist, muss er auch explizit wieder freigegeben werden. Sobald man den Rückgabewert von `add_to_array` nicht mehr braucht, benutzt man dazu die Funktion `free` (die ebenfalls in `stdlib.h` deklariert ist). Z.B. so

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

int* add_to_array (int* array, int len, int add)
{
  int i;
  int* ret;

  ret = malloc(sizeof(int) * len);

  for (i = 0; i < len; i++) ret[i] = array[i] + add;

  return ret;
}

int main (void)
{
  int i;
  int A[3] = { 1, 2, 3 };
  int *B;

  for (i = 0; i < 3; i++) printf("%d\n", A[i]);

  B = add_to_array(A, 3, 10);
  for (i = 0; i < 3; i++) printf("%d\n", B[i]);
  free(B);
```

```
  return 0;
}
```

Dabei ist zu beachten, dass man `free` nur Adressen übergeben darf, die vorher von einem Aufruf von `malloc` reserviert worden sind. Das bedeutet insbesondere, dass man einen durch `malloc` reservierten Speicherbereich nur ,,en block" wieder freigeben kann. Außerdem darf jede vom `malloc` zurückgegebene Adresse nur *einmal* freigegeben werden. Ansonsten stürzt das Programm ab.

Übung: .....

1. was wenn `malloc` keinen Platz mehr findet?
2. pass by reference vs. pass by value

.....

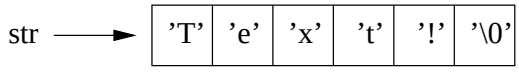
2.8.5 Strings

Eine spezielle Art von Zeigern bzw. Arrays sind sogenannte **Zeichenketten** (*engl.* ,,Strings"). Wir haben bereits in vielen Beispielen gesehen, dass der Funktion `printf` eine Zeichenkette übergeben wird, die in doppelte Anführungszeichen eingeschlossen ist. Solche, in doppelte Anführungszeichen eingeschlossene Zeichenketten, sind sogenannte String-Literale. Intern werden diese als ein Character-Array betrachtet, dessen Länge der Länge der Zeichenkette plus eins entspricht. Das eine zusätzliche Byte wird dazu verwendet, das Character-Array mit einem `NUL`-Character (ASCII-Code 0, siehe auch Tabelle 2.3) abzuschliessen. Auf diese Weise wird Funktionen wie `printf` das Ende der Zeichenkette angezeigt.

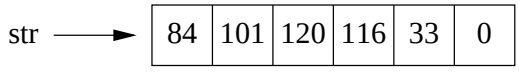
Nach dieser Erläuterung ist klar, dass

```
char* str;
...
str = "Text!";
```

eine gültige Zuweisung ist. Nach der Ausführung dieses Codefragments zeigt `str` dann auf einen Speicherbereich, in dem Folgendes steht



bzw. wenn wir statt der Buchstaben die ASCII-Codes verwenden:



Dass Strings in `C` mit einem `NUL`-Character abgeschlossen werden, hat einerseits den Vorteil, dass man nicht extra die Länge des Strings speichern muss. Andererseits ergibt sich aber zum einen das Problem, dass ein String damit niemals einen `NUL`-Character enthalten darf (denn der würde fälschlicherweise als Ende des Strings interpretiert) und man zum anderen die Länge eines Strings (wenn man sie dann einmal braucht) immer umständlich selbst ermitteln muss:

```
int length (char* str)
{
  int len = 0;
  while (str[len] != '\0') len++;
  return len;
}
```

Dieses Code-Beispiel verdeutlicht auch noch einmal, dass der abschliessende `NUL`-Character als nicht zum String gehörig betrachtet wird.

Übung: .....

1. `const`,
2. `volatile`,
3. pass by value, pass by reference,
4. Deklaration vs. Definition,
5. Header-Dateien,
6. `typedef`,
7. Casts,
8. `static`,
9. `union`

.....

Kapitel 3

Programmentwicklung

There are two ways of constructing a software design:
one way is to make it so simple that there are obviously no deficiencies;
the other is to make it so complicated that there are no obvious deficiencies.
— C. A. R. Hoare

Programming today is a race between software engineers
striving to build bigger and better idiot-proof programs,
and the universe trying to produce bigger and better idiots.
So far the universe is winning.
— Rich Cook

In den bisherigen Programmbeispielen und Übungen haben wir uns immer darauf beschränkt, allen erforderlichen Code in eine Quelldatei zu schreiben. Dies konnten wir tun, weil die zu lösenden Probleme relativ klein und überschaubar waren und wir zum anderen Funktionen wie `printf` benutzen konnten, ohne sie zu implementieren.

Die meisten Computerprogramme bestehen heute allerdings nicht aus einer, sondern aus mehreren hundert, tausend oder zehntausend Quelldateien – je nach benutzter Programmiersprache und Programm. Obwohl man in den meisten Fällen, theoretisch allen Code in eine Zeile schreiben könnte, ist es doch wesentlich einfacher, übersichtlicher, entwicklungsfreundlicher und weniger fehleranfällig, wenn man den Code über mehrere Dateien verteilt. Dabei werden ähnliche oder logisch zusammengehörige Funktionen meist in einer Datei zusammengefaßt. Das richtige und effiziente Verteilen von Code über mehrere Dateien ist ein Problem für sich und bedarf im Normalfall einiger Erfahrung, um wirklich das Optimum zu erreichen. **Hinweis für die Vorlesung:**_____ Warum ist es nicht sinnvoll, jede Funktion in eine eigene Datei zu schreiben?

In der Realität läuft die Entwicklung eines Computerprogramms meist so ab, wie in Abbildung 3.1 dargestellt:

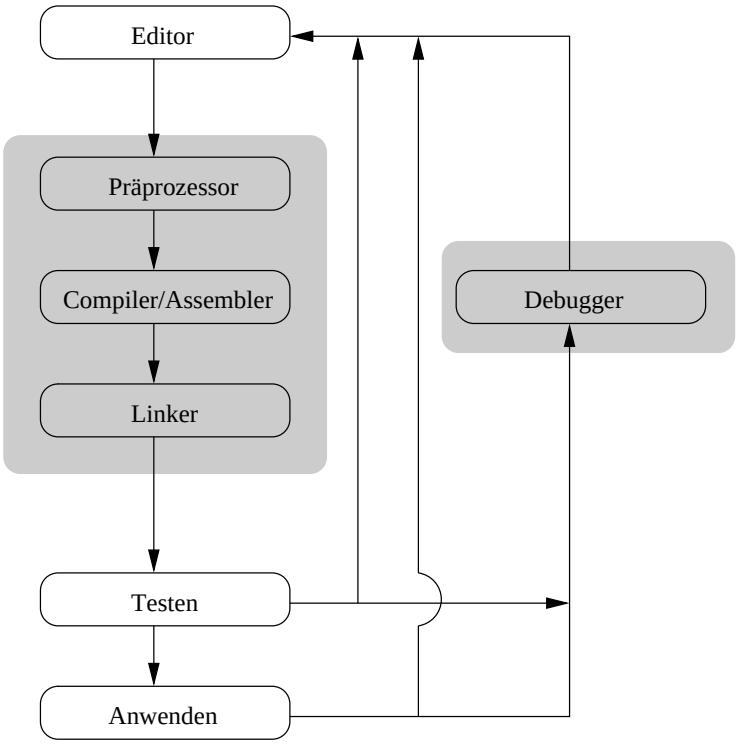


Abbildung 3.1: Tournaround-Zyklus in der Programmentwicklung.

1. Zunächst werden die erforderlichen Quelldateien erstellt (dazu benutzt man im Normalfall einen Editor).

2. Danach wird jede Quelldatei *einzeln* mittels Präprozessor, Compiler und Assembler in eine eigene *Objektdatei* übersetzt.

3. Diese Objektdateien werden dann vom Linker zu einem ausführbaren Programm zusammengebunden (daher der Name ,,Linker").

4. Das ausführbare Programm wird getestet bzw. auf das zu lösende Problem angwendet.

5. Tauchen im vorhergehenden Schritt Probleme/Fehler auf, so werden diese (unter Umständen unter Zuhilfenahme eines Debuggers) im Quellcode behoben und das Programm wird neu übersetzt.

Diese zyklische Vorgehensweise bezeichnet man auch als ,,Turnaround".

In diesem Kapitel wollen wir nun auf die in Abbildung 3.1 grau hinterlegten Teile dieses Zyklus' etwas detaillierter eingehen. Dabei können wir natürlich nicht die Detailliertheit einer Vorlesung erreichen, die sich ausschließlich mit solchen Dingen beschäftigt. Interessenten seien dazu an den Fachbereich Informatik verwiesen.

Bisher haben wir zum Übersetzen eines Quelltextes in ein Programm immer die ,,GNU Compiler Collection" (`gcc`) benutzt. Dieses Programm ist Präprozessor, Compiler, Assembler und Linker in einem, bzw. ruft die entsprechenden Unterprogramme wenn nötig auf. Wir verwenden

auch im Weiteren `gcc`, werden aber zeigen, wie man dieses Programm ausschließlich als Präprozessor, Compiler etc. einsetzen kann.

3.1 Präprozessor

Bereits in den einführenden `C`-Beispielen in Abschnitt 2.1 war angeklungen, dass Programmzeilen, die mit dem ,,Hash-Mark" (`'#'`) beginnen, vom Präprozessor bearbeitet werden. Im Wesentlichen können solche Zeilen drei für uns interessante Zwecke erfüllen

1. Andere Dateien mittels `#include` einlesen,
2. Makros durch `#define` definieren oder
3. Code mit Hilfe von `#if`, `#ifdef`, `#ifndef`, `#else` und `#endif` ausklammern.

Die Ausgabe des Präprozessors wird im Normalfall an den Compiler weitergereicht, der daraus dann den entsprechenden Objektcode generiert.

3.1.1 `#include`

Mit `#include` werden im Normalfall sogenannte ,,Header-Dateien" eingebunden. Solche Dateien enden meist auf `.h` und beinhalten eine Vielzahl von Funktions*deklarationen*, Definitionen von `struct` s und/oder `typedef` s. Trifft der Präprozessor auf eine `#include`-Anweisung, dann tut er so, als ob der Inhalt der entsprechenden Datei anstelle der `#include`-Anweisung stünde.

So ist z.B. in der Header-Datei `/usr/include/stdio.h`, die wir bereits mehrfach mit `#include <stdio.h>` eingebunden haben die Funktion `printf` deklariert. **Hinweis für die Vorlesung:**
Warum darf man den Pfad weglassen?

Diese Deklaration sieht im Wesentlichen (d.h. wenn man alle Teile, die wir bisher noch nicht gesehen haben wegläßt) so aus:

```
int printf (char* format, ...);
```

Eine solche *Deklaration* teilt dem Compiler mit, dass irgendwo in unserem Code eine Funktion `printf` definiert ist, die als erstes Argument einen String und danach eine beliebige Anzahl weiterer Argumente übernimmt und einen `int` (nämlich die Anzahl geschriebener Zeichen) zurückgibt. Im Gegensatz zu einer *Definition* wird hier nicht gesagt, wie diese Funktion nun im Einzelnen aussieht. **Hinweis für die Vorlesung:**
Unterschied Deklaration vs. Definition herausstreichen, One-Definition-Regel

---

Um einen Funktionsaufruf zu generieren, muss der Compiler das allerdings auch nicht wissen. Es reicht ihm zu wissen, welche Argumente übergeben werden und was zurückgegeben wird.

Durch das Einbinden von `stdio.h` machen wir dem Compiler also eine Vielzahl von Funktionen bekannt, die wir ab sofort benutzen können, als hätten wir sie selbst geschrieben.

Um sich dieses Einbinden zu verdeutlichen, kann man z.B. eine Header-Datei `my-header.h` schreiben

```
int my_function (int arg1, char arg2, double* arg3);
```

und diese in ein Programm `my-program.c` einbinden:

```
#include "my-header.h"

int main (void)
{
  return 0;
}
```

Dieses Programm tut zwar nichts Sinnvolles, allerdings kann man es durch den Präprozessor schicken und sich die Ausgaben ansehen. Zu diesem Zweck ruft man `gcc` mit der Option `-E` auf, also `gcc -E my-program.c`. Dies veranlaßt `gcc` dazu, das Resultat des Präprozessors auf dem Bildschirm auszugeben und danach die Bearbeitung der Eingabedateien zu beenden. In unserem Fall würde Folgendes auf dem Bildschirm ausgegeben:

```
# 1 "my-program.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "my-program.c"
# 1 "my-header.h" 1
int my_function (int arg1, char arg2, double* arg3);
# 2 "my-program.c" 2

int main (void)
{
  return 0;
}
```

Die Zeilen, die nun noch mit '`#`' beginnen, können wir ignorieren – sie dienen später dem Compiler bzw. dem Debugger um eventuelle Fehlermeldungen lesbarer zu gestalten.

Um die Ausgabe des Präprozessors mit dem Compiler weiterzubehandeln, ist es natürlich unzweckmäßig, wenn diese Ausgabe auf dem Bildschirm erfolgt. Zu diesem Zweck benutzt man die `gcc`-Option `-o`. Sie spezifiert allgemein eine Datei, in die die Ausgabe geschrieben wird. Präprozessor-Ausgaben sollten in eine Datei mit der Endung `.i` geschrieben werden, sodass das entsprechende Kommando für das ,,Hello World!"-Program aus Kapitel 2 so aussieht

```
gcc -E -o hello.i hello.c
```

Nach diesem Befehl sollten einmal die beiden Dateien `hello.c` und `hello.i` verglichen werden.

3.1.2 `#define`

Mit `#define` können sogenannte *Makros* definiert werden. Diese Makros funktionieren ähnlich wie Funktionen, werden aber bereits zur Zeit des Compilierens aufgelöst. Am leichtesten läßt sich dies durch ein Beispiel erklären. Angenommen wir haben eine Header-Datei `my-macro.h` mit dem folgenden Inhalt:

```
#define macro(_x) _x + 1

int func (int x);
```

und eine Programmdatei `my-macro.c`

```
#include "my-macro.h"

int i = 1;
int j, k;

j = macro(i);
k = func(i);
```

Dann liefert `gcc -E my-macro.c` (wenn man '`#`'-Zeilen wegläßt)

```
int func (int x);

int i = 1;
int j, k;

j = i + 1;
k = func(i);
```

In `my-header.h` haben wir ein Makro definiert, das wir – nicht besonders einfallsreich – `macro` genannt haben und dem ein Argument übergeben werden muss (den Typ des Arguments müssen wir nicht spezifizieren!). **Hinweis für die Vorlesung:** Warum braucht man keinen Typ?

Nach dieser Definition hat der Präprozessor den Text `macro(i)` durch `i + 1` ersetzt. Wir können also statt `i + 1` jetzt immer `macro(i)` schreiben.

In den meisten Fällen werden Makros dazu verwendet, um häufig auszuführende Befehle abzukürzen (so wie wir das im Code-Beispiel oben gemacht haben), oder um Konstanten zu definieren. Dies geschieht z.B. mit

```
#define PI 3.14159265358979323846
```

Nach dieser Definition des Makros `PI` (das diesmal keine Argumente hat) können wir nun überall in unserem Code `PI` schreiben und uns das langatmige 3.14159265358979323846 abkürzen. Der Präprozessor wird jedes Auftreten von `PI` durch diese Zahl ersetzen.

Eine weitere Verwendungsmöglichkeit von `#define` ist die Definition von Schaltern, die wir im nächsten Abschnitt besprechen.

3.1.3 `#if`, `#ifdef`, `#ifndef`, `#else` und `#endif`

Wie bereits oben erwähnt, dienen die Präprozessor-Direktiven `#if`, `#ifdef`, `#ifndef`, `#else` und `#endif` dazu, bei Bedarf bestimmten Code auszuklammern. Sehen wir uns dazu einmal folgendes Beispiel an:

```
#ifdef USE_SINGLE_PREC
float square_root (float f)
{
  /* compute squareroot of F and return it */
  ...
}
#else
double square_root (double d)
{
  /* compute squareroot of D and return it */
  ...
}
#endif
```

Dieses Fragment definiert auf den ersten Blick zwei Mal die Funktion `square_root`, einmal unter Verwendung einfacher und einmal unter Verwendung doppelter Genauigkeit. Ohne die Präprozessor-Direktiven würde dieser Code unweigerlich zu einem Fehler führen, denn eine Funktion darf nicht zweimal auf verschiedene Art und Weise definiert sein.

Mit den Präprozessor-Direktiven sieht die Sache aber schon etwas anders aus: Sobald der Präprozessor auf die Zeile `#ifdef USE_SINGLE_PREC` trifft, testet er, ob ein Makro namens `USE_SINGLE_PREC` definiert ist. Ist dem so, dann wird der Programmtext bis `#else` gelesen und ausgegeben, der Text zwischen `#else` und `#endif` wird dagegen ignoriert. Ist umgekehrt das Makro `USE_SINGLE_PREC` nicht definiert, dann wird der Text zwischen `#ifdef USE_SINGLE_PREC` und `#else` ignoriert, während der Text zwischen `#else` und `#endif` ausgegeben wird. Durch (Nicht-)Definition des Makros `USE_SINGLE_PREC` kann man also steuern, ob man die Wurzel mit einfacher oder doppelter Genauigkeit berechnen will. Der Vorteil hier ist, dass egal für welche Version man sich entscheidet, man im weiteren Programmverlauf immer die Funktion `square_root` verwenden kann. Insbesondere läßt sich dann durch Umdefinition von `USE_SINGLE_PREC` im *Nachhinein* die Genauigkeit der Berechnung ändern.

Ähnlich wie mit `#ifdef` im obigen Beispiel, können mit `#ifndef` und `#if` Bedingungen konstruiert werden, die darüber entscheiden, ob ein bestimmter Codeteil gelesen werden soll oder nicht.

Besonders beliebt ist es, Header-Dateien mit dem Konstrukt

```
#ifndef HEADER_H
#define HEADER_H 1

/* definition of structs */
struct xyz { ... };

/* definition of macros */
```

```
#define ...

#endif
```

vor mehrfachem Einbinden zu schützen. Da Header-Dateien selbst wieder Header-Dateien einbinden können, kommt es nämlich leicht dazu, dass eine Header-Datei mehrfach eingebunden wird. Definiert diese Header-Datei dann `struct` s oder Makros, dann führt das zu Mehrfachdefinitionen, die zwar gleich aber dennoch verboten sind. Mit dem obigen Konstrukt kann das nicht passieren:

- Wird die Datei zum ersten Mal eingebunden, dann ist das Makro `HEADER_H` nicht definiert und der Text zwischen `#ifndef HEADER_H` und `#endif` wird bearbeitet. Insbesondere wird also ein Makro namens `HEADER_H` definiert.
- Wird die Datei ein weiteres Mal eingelesen, dann ist das Makro `HEADER_H` bereits definiert, d.h. der Text zwischen `#ifndef HEADER_H` und `#endif` wird ignoriert. Das hat den gleichen Effekt, als ob die Datei gar nicht eingelesen wurde.

Selbstverständlich muss das Makro, das wir hier `HEADER_H` genannt haben, für jede Header-Datei einen anderen Namen haben. Üblicherweise wählt man als Makro-Namen den namen der Header-Datei, wobei Klein- durch Großbuchstaben ersetzt werden. Sieht man sich z.B. Mal ein paar Header-Dateien unter `/usr/include` an, dann wird man dieses Prinzip in nahezu jeder Datei wiederfinden.

Übung:

1. Was wird inkludiert? `gcc -E -dI hello.c | grep '#include'`
2. Ein paar Makros schreiben, ein bischen mit `#ifdef` experimentieren

3.2 Compiler/Assembler

Nachdem der Präprozessor seine Arbeit beendet und den resultierenden Programmtext an den Compiler weitergegeben hat, ist es Aufgabe des Compilers aus diesem Text nun den entsprechenden *Objektcode* zu generieren.

Dieser Objektcode ist im Wesentlichen bereits ein Programmfragment in Maschinensprache. Allerdings enthält der Objektcode noch einige Dinge, die später vom Linker bereinigt werden müssen (siehe auch Abschnitt 3.3 weiter unten). Um aus der Datei `hello.i` des vorherigen Abschnitts den entsprechenden Objektcode zu erzeugen, rufen wir

```
gcc -c -o hello.o hello.i
```

auf. Der Parameter `-c` sorgt dabei dafür, dass kein ausführbares Programm, sondern lediglich die Objektdatei erzeugt wird, während mit `-o` wieder die Ausgabedatei angegeben wird. Für Objektdateien sollte man möglichst die Endung `.o` wählen. Leider kann man die Objektdatei nun nicht mehr mit der Quelldatei `hello.c` oder der Eingabedatei `hello.i` vergleichen, da `hello.o` bereits zu grossen Teilen aus Maschinencode besteht. Diesen Code sieht man sich mit dem Programm `objdump` ansehen:

```
objdump -D hello.o
```

Dabei stören wir uns nicht daran, dass uns das alles nichts sagt, schließlich sind wir erst am Beginn unserer Programmierkarriere.
Mit

```
gcc -S -o hello.s hello.i
```

kann man sich übrigens eine Zwischenstufe, das sogenannte ,,Assembler-Listing" erzeugen lassen. In diesem Listing kann man immerhin noch die Begriffe ,,main" und ,,printf" ausmachen sowie den String ,,Hello World!\n" sehen.

3.3 Linker

Nachdem wir nun verschiedenen Quelldateien in ihre jeweiligen Objektdateien übersetzt haben, bleibt die Frage: wie wird aus diesen Objektdateien ein ausführbares Programm? Wie bereits oben erwähnt übernimmt diese Aufgabe der Linker.

Dabei kümmert sich der Linker insbesondere um Folgendes

- Das Zusammenschreiben der Objektdateien kann nicht ,,irgendwie" geschehen, sondern muss nach einem bestimmten System erfolgen. Der Linker stellt sicher, dass dieses System eingehalten wird.
- Beim Übersetzen von Quelldateien waren möglicherweise teilweise nur Funktionsdeklararionen und keine Funktionsdefinitionen verfügbar. Überall wo dies der Fall war, muss der Linker die Objektdatei suchen, die die Funktion tatsächlich definiert und in den anderen Quelldateien die entsprechende Adresse einfügen. Definiert keine Datei die entsprechende Funktion, so muss der Linker einen Fehler melden.
- Einige Funktionen (wie z.B. `printf`) werden nicht in Quelldateien definiert, die der Benutzer geschrieben hat, sondern in sogenannten Bibliotheken. Bevor der Linker also eine Funktion als nicht definiert bemängelt, durchsucht er bestimmte Bibliotheken, ob die Funktion nicht in einer von ihnen definiert ist. Ist dem so, dann muss die entsprechende Bibliothek zusammen mit den Objektdateien in die ausführbare Datei gelinkt werden. Dabei muss insbesondere darauf geachtet werden, dass eine Bibliothek nicht mehrfach in der ausführbaren Datei landet.

Nun ist also klar, woher die Funktion `printf` in unseren ganzen bisherigen Code-Beispielen kam: der Linker hat sie aus einer entsprechenden Bibliothek entnommen. Diese Bibliotheken sind ein wesentliches Konzept, um den Programmierer davor zu bewahren, die gleiche Funktionalität wieder und wieder implementieren zu müssen.

Um aus der Objektdatei `hello.o` des vorherigen Abschnitts nun endlich das ausführbare Programm zu erstellen, rufen wir

```
gcc -o hello hello.o
```

auf. Um aus mehreren Objektdateien ein ausführbares Programm zu erstellen, müssten wir einfach statt `hello.o` die Liste aller Objektdateien angeben, die wir linken wollen.

Für Fortgeschrittene: .....

- statisches/dynamische Linken
- `extern`, `static`
- Name (de)mangling (Linker hängt nicht mehr von der Programmiersprache ab!)

.....

3.4 make

Obwohl uns der `gcc` bereits die meisten der Schritte von oben abnimmt, ist die Arbeit mit vielen Quelldateien in grossen Software-Projekten immer noch mühsam. Um uns diese Arbeit zu erleichtern, benutzen wir das Program `make`.

Grundsätzlich gelten für ein Programm die folgenden Abhängigkeiten:

1. Sobald sich eine Quelldatei geändert hat, muss die entsprechende Objektdatei neu erzeugt werden.
2. Hat sich eine Datei A geändert, die von Datei B mittels `#include` (direkt oder indirekt) eingebunden wird, so muss aus der Datei B eine neue Objektdatei erzeugt werden.
3. Sobald sich eine Objektdatei geändert hat, muss die ausführbare Datei neu erzeugt werden.

Hat man mehrere tausend Quelldateien, dann wird es sehr schnell sehr schwierig, nach einer Änderung an einer Quelldatei herauszufinden, welche Objektdateien neu erzeugt werden müssen. Um diese Arbeit zu erleichtern, wurde das Programm `make` entwickelt. Anhand einer Steuerdatei – dem sogenannten ,,Makefile" – bestimmt das Programm die neu zu erzeugenden Objektdateien und generiert diese sowie das auszuführende Programm.

Eine detaillierte Einführung in `make` würde den Rahmen dieser Vorlesung mehr als sprengen. Deshalb beschränken wir uns hier darauf, eine Schablone für ein Makefile vorzustellen, die für die Anwendungen in dieser Veranstaltung ausreichend ist. Dabei gehen wir davon aus, dass wir drei Quelldateien `source-1.c`, `source-2.c` und `source-3.c` haben, die zusammen das ausführbare Program `my-program` ergeben sollen. Zu diesem Zweck benutzen wir die folgenden Steuerdatei:

```
# compiler and linker
CC      = gcc
CFLAGS  = -W -Wall
LDFLAGS =

# source files, object files, dependency files, program
sources = source-1 source-2 source-3
objects = $(addsuffix .o, $(sources))
depends = $(addsuffix .d, $(sources))
```

```
program = my-program

# dependencies and rules
all: $(program)

$(program): $(objects)
        $(CC) -o $(program) $(objects) $(LDFLAGS)

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $<

%.d: %.c
        $(CC) $(CFLAGS) -M $< > $@

-include $(depends)

clean:
        rm -f $(objects)

distclean: clean
        rm -f $(depends)
```

Wenn wir diese Datei im gleichen Verzeichnis wie `source-1.c`, `source-2.c` und `source-3.c` unter dem Namen `Makefile` abspeichern, dann können wir mit

```
make my-program
```

oder einfach mit

```
make
```

die ausführbare Datei `my-program` erzeugen bzw. aktualisieren. Mit

```
make clean
```

bzw.

```
make distclean
```

können wir die Dateien löschen, die auf dem Weg zur ausführbaren Datei erzeugt werden und nicht mehr benötigt werden, sobald die ausführbare Datei aktualisiert ist.

Die oben angegebene Steuerdatei besteht aus drei Sektionen, die wir kurz erläutern wollen:

Compiler und Linker Hier werden einige Variablen definiert, die festlegen, welchen Compiler bzw. Linker wir benutzen wollen und mit welchen Argumenten (Flags) diese Programme aufzurufen sind. Wir benutzen `gcc` sowohl als Compiler als auch als Linker. Wenn `gcc` als Compiler benutzt wird, soll das mit den Flags `-W` und `-Wall` geschehen. Wenn `gcc` als Linker verwendet wird, dann sind keine weiteren Flags erforderlich.

In dieser Sektion sehen wir bereits, wie in einer `make`-Steuerdatei Variablen definiert werden: Man schreibt einfach „Variable = Wert" um eine Variable zu definieren und gleichzeitig zu initialisieren. `make` speichert alle Variablen als Strings, deshalb ist auch die Angabe eines Variablentyps nicht notwendig. Außerdem sehen wir, dass wir Leerzeichen diesmal nicht „quoten" müssen.

Dateien In dieser Sektion haben wir verschiedene Dateilisten definiert. Zunächst ist dies die Liste `sources` der Quelldateien (bei denen wir der Einfachheit halber das Suffix `.c` weggelassen haben).

Danach folgt die Liste `objects` der Objektdateien. Diese entstehen aus den Quelldateien jeweils durch Anhängen von `.o`. Genau dies bewirkt `$(addsuffix ...)`. In unserem Fall ist die Liste der Objektdateien also `source-1.o`, `source-2.o` und `source-3.o`.

Analog zu den Objektdateien haben wir eine Liste von Abhängigkeitsdateien definiert. Auf diese Dateien wollen wir nicht näher eingehen, sondern lediglich erwähnen, dass diese Dateien dazu dienen, die Abhängigkeiten zu beschreiben, die wir oben als zweiten Punkt aufgelistet haben.

Schließlich haben wir noch den Namen unseres Programms festgelegt.

In dieser Sektion sehen wir, wie man in einer `make`-Steuerdatei Variablen referenziert und Funktionen aufruft: beides geschieht mit `$(...)`. Den Inhalt der Variablen `sources` erhalten wir mit `$(sources)` und die Funktion `addsuffix` rufen wir mit `$(addsuffix ...)` auf. Die Argumente werden der Funktion einfach als durch Komma getrennte Liste von Strings übergeben.

Abhängigkeiten und Regeln Hier haben wir festgelegt, von welchen Dateien (nämlich den Objektdateien) das ausführbare Programm abhängt und wie es aus diesem Dateien zu erstellen ist.

Danach haben wir angegeben, wie eine Datei mit der Endung `.o` aus einer entsprechenden Datei mit der Endung `.c` gewonnen werden kann, nämlich durch einen entsprechenden Aufruf des Compilers.

Als nächstes generieren wir die Abhängigkeitsdateien und binden diese an.

Schlussendlich haben wir noch festgelegt, was passieren soll, wenn `make clean` oder `make distclean` aufgerufen wird.

Obwohl man mit der hier vorgestellten Schablone schon sehr weit kommen kann, empfehlen wir doch jedem, sich im Laufe der Zeit die durch `info make` zur Verfügung gestellte Dokumentation des Programms anzueignen. In der Praxis zeigt sich nämlich immer wieder, dass sich durch richtig/geschickt geschriebene `make`-Steuerdateien viel Arbeit und Mühe ersparen läßt und häufig gemachte (und schwer aufzuspürende) Fehler vermieden werden können.

Übung:

1. Makefile und source-?.[hc] runterladen, Programm builden.
2. Grobe Idee bekommen, was Makefile tut:
   (a) CFLAGS ändern,

   (b) LDFLAGS ändern,
   (c) sources ändern,
   (d) Programmname ändern.
   (e) Bibliothek benutzen

3. Eigenes Programm aus mehreren Quellen erstellen (gutes Beispiel?)

.....

3.5 Debugging

Im Gegensatz zu Syntax-Fehlern (vgl. Abschnitt 2.2), die bereits beim Übersetzen des Programms entdeckt werden können, sind Semantik-Fehler meist wesentlich schwerer aufzuspüren. Während ein Syntax-Fehler dazu führt, dass das Programm schlicht nicht übersetzt werden kann, führt ein Semantik-Fehler dazu, dass das Programm zwar übersetzt wird, aber nicht das tut, was es tun soll. Im besten Fall stürzt das Programm einfach ab. Es kann aber auch durchaus Schlimmeres passieren: z.B. könnte das Programm wegen eines Semantik-Fehlers nicht nur eine Datei, sondern die gesamte Festplatte löschen. Oder das Programm sperrt während seiner Ausführung den Drucker für andere Benutzer und gibt diesen nicht mehr frei.

Eine der häufigen Ursachen für semantische Fehler ist sogenanntes ,,undefiniertes Verhalten": in (fast) allen Programmiersprachen gibt es Aktionen, die man durchführen kann, deren Ergebnis aber undefiniert ist. Beispiele für solche Aktionen sind:

1. Auslesen einer nicht-initialisierten Variable,
2. Auslesen von oder allgemeiner Zugriff auf nicht-reservierten Speicher,
3. Zugriff auf ein Arrayelement `A[n]` wobei `n` entweder kleiner als 0 oder größer gleich der Länge des Arrays ist,
4. Freigeben von Speicher durch `free()`, der nicht durch `malloc()` oder `realloc()` reserviert wurde,
5. mehrfaches Freigeben von Speicher mittels `free()`, der durch `malloc()` oder `realloc()` reserviert wurde,

Theoretisch ließen sich alle der hier aufgeführten Beispiele zur Laufzeit des Programms automatisch erkennen. Das würde allerdings zu sehr viel größeren und langsameren Programmen führen, sodass man in diesen Fällen auf diese Tests zur Laufzeit verzichtet hat. Stattdessen ist es Aufgabe des Programmierers, dafür zu sorgen, dass diese Fälle nicht eintreten.

Sollte es doch einmal zu semantischen Fehlern kommen (und das passiert häufiger als einem lieb ist), so hilft in vielen Fällen ein *Debugger*. Ein Debugger dient dazu, ein anderes Programm auszuführen und während der Ausführung zu überwachen. So kann man mit einem Debugger z.B.

- ein Programm Schritt für Schritt ausführen,
- sich während der Ausführung die Werte von Variablen oder Speicherbereichen anzeigen lassen,

- das Programm an vorher definierten Stellen anhalten,
- während der Laufzeit Variablen ändern,
- und vieles mehr.

In der Übung werden wir uns einen dieser Debugger, den GNU-Debugger `gdb`, etwas genauer ansehen.

Kapitel 4

Abstrakte Datentypen

Abstraction is selective ignorance.
— Andrew Koenig

In Kapitel 2 hatten wir bereits gesehen, wie man mit Hilfe des Schlüsselwortes `struct` oder mit Hilfe von Arrays eigene Datentypen definieren kann. Bisher dienten Datentypen, die wir so definiert haben, immer einem konkreten Zweck (z.B. der Behandlung von komplexen Zahlen mit `struct complex`). Aus diesem Grund werden solche Datentypen auch als **konkrete Datentypen** (*engl.* ,,concrete datatypes") bezeichnet.

In diesem Kapitel wollen wir uns nun den **abstrakten Datentypen** (*engl.* ,,abstract datatypes" oder ADTs) widmen. Hierbei handelt es sich um Datentypen, die nicht für einen bestimmten Zweck entworfen wurden, sondern vielfach und variabel einsetzbar sind. Viele dieser Datentypen ließen sich z.B. mit Hilfe eines Arrays implementieren, sofern man die Menge der anfallenden Daten *a priori* kennt. Kennt man diese Menge nicht, so wird die Implementierung mit Arrays sehr ineffizient und aufwändig, während die Implementierung mit ADTs nach wie vor einfach zu bewerkstelligen ist.

Bevor wir genauer auf diese abstrakten Datentypen eingehen, müssen wir zunächst ein weiteres `C`-Konzept besprechen, das hier Anwendung findet: die **untypisierten Zeiger**.

4.1 Untypisierte Zeiger: `void*`

In Kapitel 2 haben wir bereits die typisierten Zeiger kennengelernt. Dies waren Variablen des Typs `char*`, `int*`, `char**` usw. Wie wir wissen, enthält ein typisierter Zeiger eine Speicheradresse, an der eine Variable des gegebenen Typs reserviert ist.

Im Gegensatz zu den typisierten Zeigern gibt es in `C` auch noch die **untypisierten Zeiger**. Untypisierte Zeiger werden durch `void*` deklariert und sind mit allen typisierten Zeigertypen ,,kompatibel", d.h. ein Zeiger eines beliebigen Typs kann einem entsprechenden untypisierten Zeiger zugewiesen werden und umgekehrt. So sind z.B. die Zuweisungen in

```
char* string;
void* void_ptr;
```

```
int*  int_ptr;

string = "This is a string";
void_ptr = string;
int_ptr = void_ptr;
```

zulässig (auch wenn sie nicht besonders sinnvoll sind), wohingegen die direkte Zuweisung `int_ptr = string` nicht erlaubt ist (siehe auch Abschnitte 2.3.3 und 2.8.2). Im Gegensatz zu typisierten Zeigern können untypiserte Zeiger allerdings nicht mittels `*` oder `->` dereferenziert werden: es ist ja gar nicht klar, was für ein Typ derefenziert werden soll.

Eine Anwendung von untypisierten Zeigern haben wir unter der Hand schon kennengelernt: die Funktionen `malloc` und `free` sind wie folgt deklariert.

```
void* malloc (size_t size);
void free (void* ptr);
```

Wie wir bereits wissen, reserviert ein Aufruf von `malloc` genau `size` Bytes im Arbeitsspeicher und gibt einen Zeiger auf diesen reservierten Bereich zurück. Da die Funktion `malloc` aber natürlich nicht wissen kann, von welchem Typ die Variablen sein werden, die der Programmierer in diesem Bereich ablegen will, kann `malloc` auch nicht den passenden Zeigertyp zurückgeben. Selbst wenn die Funktion das könnte, wäre es sehr unpraktisch, denn dann müßte man für jeden Variablentyp eine eigene `malloc`-Funktion schreiben. Stattdessen gibt `malloc` eben einen untypisierten Zeiger zurück, den man dann bequem jedem typisierten Zeiger zuweisen kann.

Genau umgekehrt verhält es sich bei `free`: auch hier kann die Funktion natürlich nicht wissen, welche Variablen der Programmierer jetzt in dem von `malloc` reservierten Bereich abgelegt hat. Zum Freigeben des reservierten Bereichs muss die Funktion aber andererseits nur wissen, wo denn dieser Bereich nun ist. Also wird als Parameter-Typ ein untypisierter Zeiger verwendet. Das hat zur Folge, dass man `free` mit jedem typisierten Zeigertyp aufrufen kann. Beim Aufruf wird der typisierte Zeiger dann einfach implizit in einen untypisierten Zeiger umgewandelt.

In abstrakten Datentypen werden untypisierte Zeiger normalerweise wie folgt verwendet: Für einen abstrakten Datentyp ADT gibt es im Regelfall zwei Funktionen der Bauart

```
void adt_insert (ADT adt, void* elem);
void* adt_extract (ADT adt);
```

Die Funktion `adt_insert` fügt ein neues Element (`elem`) in `adt` ein und `adt_extract` entfernt ein bestimmtes Element aus `adt` und gibt diese Element zurück. Dabei sind alle Elemente vom Typ `void*`, also untypisierte Zeiger. Auf diese Art muss man den abstrakten Datentypen nur einmal programmieren und kann ihn sofort für alle Arten von Zeigern benutzen. Den fiktiven Datentyp ADT könnte man z.B. wie folgt verwenden:

```
ADT adt;
int i;
int* int_ptr;

/* insert elements into adt */
for (i = 0; i < 5; i++) {
  int_ptr = malloc(sizeof(int));
```

```
  *int_ptr = i;
  adt_insert(adt, int_ptr);
}

/* extract elements from adt */
for (i = 0; i < 5; i++) {
  int_ptr = adt_extract(adt);
  printf("%d\n", *int_ptr);
  free(int_ptr);
}
```

Wenn ADT jetzt z.B. die Elemente in umgekehrter Einfügereihenfolge ausgibt, so werden auf der Konsole nacheinander die Zahlen 4, 3, 2, 1 und 0 ausgegeben.

4.2 Nullzeiger

Im Prinzip ist es erlaubt, jedem Zeiger eine beliebige Zahl zuzuweisen:

```
char* ptr = ...;
...
ptr = 5;
```

Führt man eine solche Zuweisung aus, dann wird in `ptr` eben einfach die Adresse 5 gespeichert. Im Normalfall sind solche Zuweisungen weder sinnvoll noch hilfreich. Es gibt allerdings eine Ausnahme: die Zuweisung von 0 an einen Zeiger. Die Adresse 0 ist eine Adresse, die ein Benutzer-Programm niemals lesen oder beschreiben darf. Deswegen kann man z.B. bei einem Zeiger, der die Adresse 0 enthält, absolut sicher sein, dass er bisher noch nicht mit `malloc()` oder ähnlichen Funktionen oder dem ,,`&`"-Operator initialisiert wurde. Zusätzlich ergibt die Adresse 0 den Wert ,,falsch", wenn sie als logischer Ausdruck ausgewertet wird. Im Gegensatz dazu liefern alle anderen Adressen bei einer solchen Auswertung den Wert ,,wahr".

Aus diesen beiden Gründen ist es sehr beliebt, nicht initialisierte Zeiger mit 0 zu initialisieren und vor der weiteren Verwendung dann zu überprüfen, ob sie überhaupt eine gültige Adresse enthalten:

```
char* ptr = 0;
...
if (! ptr) { ptr = malloc(...); }
ptr[0] = 'A';
...
```

Im Folgenden werden wir es neben den untypisierten Zeiger häufig auch mit dem Nullzeiger zu tun bekommen.

4.3 Stacks

Einen **Stack** (im Deutschen auch **Stapel** oder **Keller**) genannt, kann man sich vorstellen, wie einen Stapel schmutziger Teller, der von Hand gespült wird:

- Wenn ein Teller fertiggespült ist, wird der nächste schmutzige Teller von oben vom Tellerstapel genommen.

- Tauchen neue schmutzige Teller auf, so werden diese einfach oben auf den Stapel mit schmutzigen Tellern gelegt (sie drunter zu legen wäre weitaus komplizierter).

So wird also immer der Teller gespült, der als *letzter* auf den Stapel gelegt wurde. Dieses Prinzip ist in der Computerwelt unter dem Namen ,,Last In, First Out" (LIFO) bekannt.

Für uns ist ein Stack nichts wesentlich anderes als der Tellerstapel, mit dem einzigen Unterschied, dass wir in unserem Stapel keine Teller haben, sondern beliebige Elemente, die durch untypisierte Zeiger (`void*`, siehe Abschnitt 4.1) gegeben sind. Genauer gesagt, ist ein Stack eine Datenstruktur, für die die folgenden Operationen definiert sind:

push Legt ein Element oben auf den Stack.

pop Nimmt das oberste Element vom Stack und liefert es zurück.

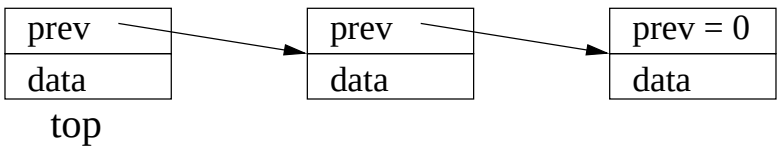
top Liefert das oberste Element des Stacks zurück (ohne es vom Stack zu nehmen).

is_empty Prüft, ob der Stack leer ist. Wenn ja, wird ein wahrer Wert, anderenfalls ein falscher Wert zurückgegeben.

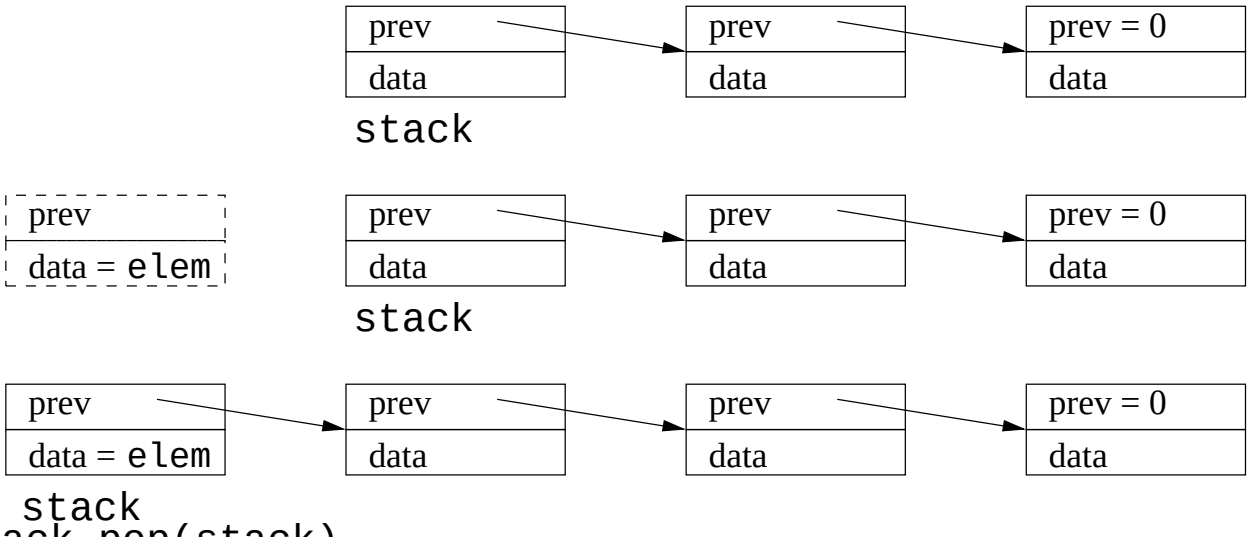
Implementiert wird ein solcher Stack normalerweise durch sogenannte **Nodes** (*deut.* ,,Knoten"): Ein Node beinhaltet zum Einen einen Zeiger auf das Element, das in diesem Node gespeichert ist, und zum anderen einen Zeiger auf den vorhergehenden (d.h. den darunterliegende) Node. So braucht man immer lediglich den obersten Node zu kennen, alle anderen lassen sich durch die jeweiligen ,,Vorgänger-Zeiger" ermitteln. **Hinweis für die Vorlesung:**

- Bilder von Nodes malen,

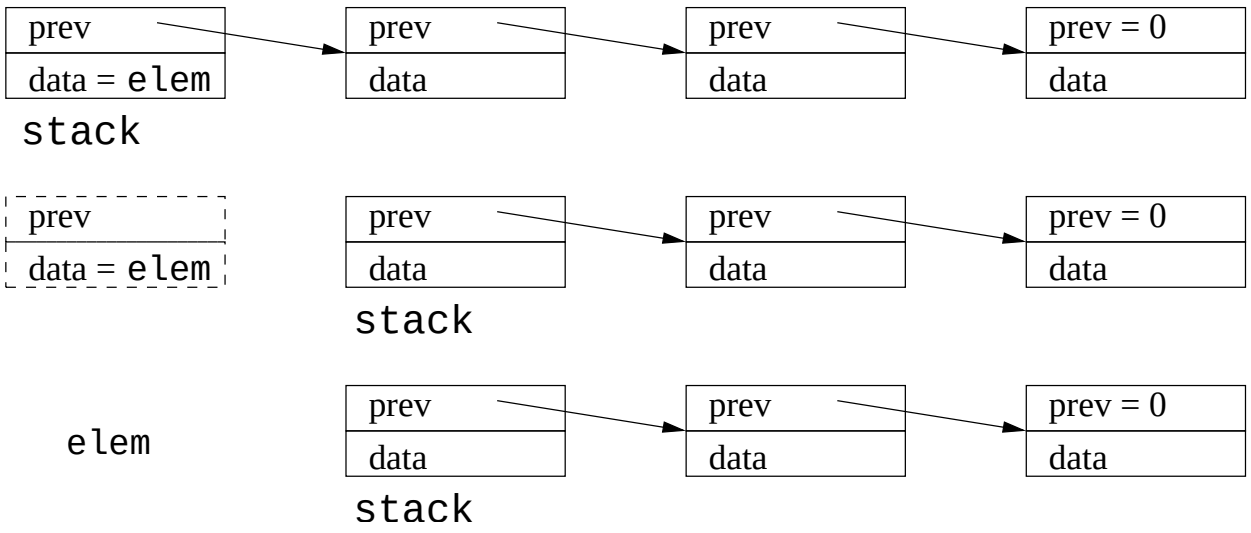
- an Bildern erklären, wie push(), pop(), top(), is_empty() funktionieren.



stack_push(stack, elem)



stack_pop(stack)



In C können wir einen solchen Node einfach mit folgender `struct` implementieren:

```
struct stack_node {
  void*                data;
  struct stack_node*   prev;
};
```

Dabei verdient das Feld `prev` besondere Beachtung: Obwohl zum Zeitpunkt, an dem `prev` definiert wird, die Definition von `struct stack_node` noch gar nicht abgeschlossen ist, können wir trotzdem einen Zeiger auf eine solche `struct` definieren. Diese Möglichkeit werden wir bei fast allen abstrakten Datentypen in diesem Kapitel ausgiebig nutzen.

Mit dieser Definition von `struct stack_node` können wir nun die benötigten Operationen definieren:

```
struct stack_node {
  void*                data;
  struct stack_node*   prev;
};

typedef struct stack_node* Stack;

#define STACK_INIT ((Stack) 0)

#include <stdlib.h>  /* for malloc() and free() */
#include <assert.h>  /* for assert() */

void stack_push (Stack* stack_ptr, void* data)
{
  /* allocate a new node */
  Stack new_node = malloc(sizeof(struct stack_node));

  /* initalise the new node */
  new_node->prev = *stack_ptr;
  new_node->data = data;

  /* let STACK_PTR point to the newly allocated node */
  *stack_ptr = new_node;
}

void* stack_pop (Stack* stack_ptr)
{
  void* ret;
  Stack prev;
  assert(stack_ptr);

  /* save DATA for return value */
  ret = (*stack_ptr)->data;
  /* save pointer to previous node */
  prev = (*stack_ptr)->prev;

  /* release top node */
  free(*stack_ptr);
  *stack_ptr = prev;

  return ret;
}
```

```
void* stack_top (Stack stack)
{
  assert(stack);

  return stack->data;
}

int stack_is_empty (Stack stack)
{
  if (stack == STACK_INIT) return 1;

  return 0;
}
```

Hinweis für die Vorlesung:
Es muss detailliert erläutert werden, wie das hier alles funktioniert. Es muss außerdem ein Beispiel gegeben werden, wie man diesen Stack jetzt benutzt.

In einem realen Programm würde man den obigen Code natürlich in einen Interface- und einen Implementierungsteil trennen, damit man den Stack auch leicht in anderen Programmen benutzen kann.

Übung:

1. Stack implementieren mit .h- und .c-File (kann alles aus Skript abgeschrieben werden).
2. ,,Stack-Machine" implementieren (nur Addition und unäres Minus) mit Hilfe von `argc, argv` (die müssen dazu in der Übung eingeführt werden)
3. Wie könnte man Stacks mit Arrays implementieren
   (a) wenn obere Schranken für Menge der Daten bekannt ist,
   (b) wenn keine obere Schranke für diese Menge bekannt ist.

   Jeweils Pro und Contra diskutieren.

4.4 Queue

Der abstrakte Datentyp **Queue** (*deut.* ,,Warteschlange") kann am besten mit einer Warteschlange an einer Kaufhauskasse verglichen werden: Als nächstes wird immer derjenige ,,bearbeitet" (bedient), der bereits am längsten in der Schlange steht. Im Gegensatz zum Stack aus Abschnitt 4.3 wird hier also nach dem ,,First In, First Out"-Prinzip (FIFO) gearbeitet.

Auch eine Queue wird wieder mit Hilfe von Nodes implementiert, allerdings braucht man hier in dem Node keinen Zeiger auf den vorhergehenden Node, sondern einen Zeiger auf den nächste Noden. Zusätzlich muss man nicht nur ein Ende der Queue kennen (beim Stack hatte es

gereicht, den obersten Node zu kennen), sondern beide. Warum dies so ist, wird klar, wenn man die Operationen betrachtet, die eine Queue unterstützen muss:

enqueue Hängt ein Element an das Ende der Queue an.

dequeue Nimmt das vorderste Element aus der Queue und gibt es zurück.

front Gibt das vorderste Element der Queue zurück (ohne es aus der Queue zu nehmen).

is_empty Gibt wahr zurück, falls die Queue leer ist und falsch andernfalls.

Die Operationen zeigen, dass man sowohl den Anfang als auch das Ende der Queue verändern können muss. **Hinweis für die Vorlesung:**
Wieder Beispiel mit Bildern, wie Operationen funktionieren

Wie man eine Queue nun genau implementiert, werden wir in der Übung sehen.

In der Praxis werden Queue-Datenstrukturen meistens dann verwendet, wenn man irgend eine Art von Warteschlangen implementieren möchte. Ein Beispiel für solche Warteschlangen ist die Drucker-Warteschlange. Wenn wir uns die `man`-Page von `lpr` ansehen, werden wir feststellen, dass dort immer wieder von einer Queue die Rede ist. Druckt man mit `lpr` wird der Druckauftrag nämlich nicht direkt ausgeführt, sondern an das Ende der Druckerwarteschlange gestellt. Erst wenn alle vorherigen Druckaufträge bearbeitet (oder abgebrochen) wurden, wird dann der angestellte Auftrag ausgeführt.

Übung: .....

1. Queue implementieren (`struct` s vorgeben?)
2. Queue mit Array implementieren? (Circular Array?)

.....

4.5 Listen

Wir haben bereits zwei spezielle Arten von Listen kennengelernt: der Stack war eine Liste, bei der wir uns nur für das jeweils erste (oberste) Element interessiert haben, während die Queue eine Liste war, wo wir uns für das jeweils erste Element und den jeweile letzten Node interessiert haben.

In dieser Veranstaltung bezeichnen wir mit *Liste* immer eine sogenannte **Doppelt Verkettete Liste** (*engl.* „doubly linked list"). Dies ist eine Datenstruktur, bei der theoretisch jedes Element und jeder Node von Interesse sind. Zu diesem Zweck, haben die Nodes in einer Doubly Linked List neben dem `data`-Feld noch *zwei* zusätzlich Felder: einen Zeiger auf den vorhergehenden und einen Zeiger auf den nachfolgenden Node. Diese zwei Zeiger ermöglichen es, die Liste sowohl vorwärts als auch rückwärts zu bearbeiten.

Listen werden insbesondere dann eingesetzt, wenn man eine unbekannte Zahl von Objekten hat, über die man später iterieren muss. Außerdem bieten Listen im Gegensatz zu Stacks und Queues auch die Möglichkeit, Elemente *in der Mitte* der Liste einzufügen und nicht nur am Anfang oder am Schluß. Normalerweise unterstützen Listen die folgenden Operationen:

insert_after Fügt ein Element nach einem angegebenen Node ein.

insert_before Fügt ein Element vor einem angegebenen Node ein.

remove_node Nimmt einen Node aus der Liste.

next_node Liefert den Nachfolger zu einem gegebenen Node.

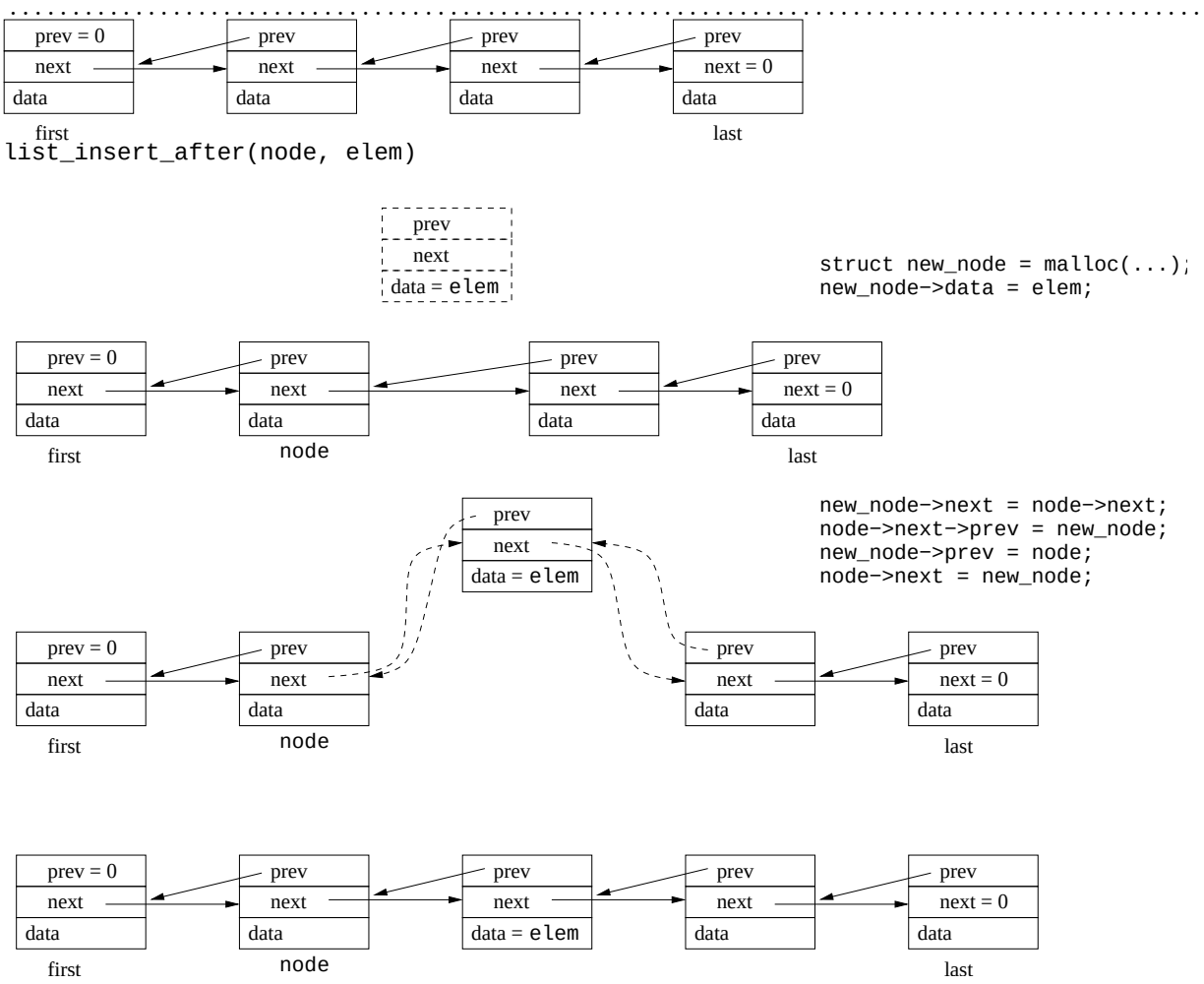
prev_node Liefert den Vorgänger zu einem gegebenen Node.

clear Löscht alle Nodes in der Liste.

Hinweis für die Vorlesung:
Wieder Beispiel mit Bildern.

Übung:

1. Liste implementieren,
2. Liste mit Array implementieren



4.6 Bäume

Den abstrakten Datentyp **Baum** (*engl.* ,,tree") kennen wir eigentlich schon aus dem Einführungskapitel: das Dateisystem ist genau so ein Baum. Ein Baum besteht – wie die anderen abstrakten Datentypen bisher auch – aus Knoten, die hier allerdings nicht mehr in einer Reihe hintereinander stehen, sondern eine Baumstruktur bilden. Ein ausgezeichneter Knoten bildet die **Wurzel** (*engl.* ,,root") des Baumes. Die Wurzel hat wiederum eine beliebige (endliche) Anzahl von **Kind-Knoten** (*engl.* ,,child nodes"). Jeder dieser Knoten hat wiederum eine beliebige (endliche) Anzahl von Kindern usw. Knoten, die mindestens ein Kind haben, werden als **interne Knoten** (*engl.* ,,internal nodes") bezeichnet. Knoten ohne Kinder heißen **externe Knoten** (*engl.* ,,external nodes") oder **Blätter** (*engl.* ,,leaves"). Im Dateibaum entsprechen die internen Knoten den nichtleeren Verzeichnissen und die Blätter den leeren Verzeichnissen und den Dateien (betrachtet man ,,`.`" oder ,,`..`" ebenfalls als Verzeichniseinträge dann sind die internen Knoten die Verzeichnisse und die Blätter die Dateien).

In der Informatik gibt es eine Vielzahl verschiedener Bäume, die für die unterschiedlichsten Zwecke eingesetzt werden können. In dieser Veranstaltung wollen wir uns auf einen speziellen Typ beschränken, den **Binären Baum** (*engl.* ,,binary tree"). Bei diesem speziellen Typ von Baum hat jeder Knoten höchstens zwei Kinder – man spricht dann auch von dem rechten und dem linken Kind. Zusätzlich hat jeder Knoten das altbekannte Feld `void* data`, mit dessen Hilfe wir an dem Knoten Daten speichern können. Streng genommen bezeichnet man als Binären Baum normalerweise Bäume, bei denen jeder interne Knoten genau zwei Kinder hat. An den Kindern, die man nicht braucht, speichert man dann einfach keine Daten (`data = 0`).

Malt man sich einen Baum wie in Abbildung 4.1 mit der Wurzel nach oben auf, dann hat

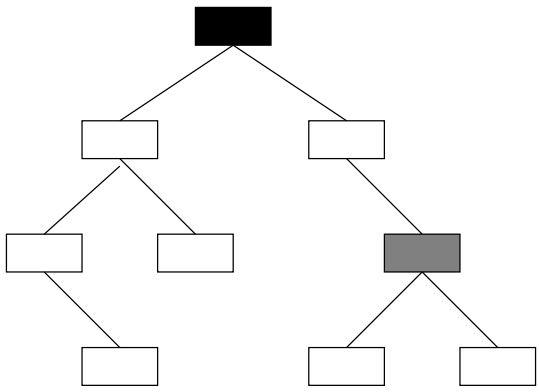


Abbildung 4.1: Ein Beispiel für einen Binären Baum.

jeder Knoten außer der Wurzel einen **Elternknoten** (*engl.* ,,parent node"). Zwei Knoten, die den gleichen Elternknoten haben bezeichnet man als **Geschwister** (*engl.* ,,siblings"). Von jedem Knoten gibt es einen (kürzesten) Weg zum Wurzelknoten. Die Anzahl der Knoten auf diesem Weg, die ungleich der Wurzel sind, bezeichnet man als die **Tiefe** (*engl.* ,,depth") oder den Level eines Knoten. Die **Höhe** (*engl.* ,,height") eines Baumes ist die Tiefe seines tiefsten Knoten.

Der Baum in Abbildung 4.1 hat also eine Höhe von 3. Der schwarze Knoten ist die Wurzel, der graue Knoten ist ein interner Knoten ohne Geschwister und hat die Tiefe 2.

Übung: .....

1. arithmetischen Ausdruck durch einen binären Baum darstellen: interne Knoten = Operator, Blätter = Zahlen

.....

In `C` implementieren wir einen Binären Baum einfach mit der folgenden `struct`:

```
struct tree_node {
  void*              data;

  struct tree_node*  parent;
  struct tree_node*  left;
  struct tree_node*  right;
};
```

Die Verkettung der einzelnen Knoten eines Binären Baumes erfolgt dann wie in Abbildung 4.2.

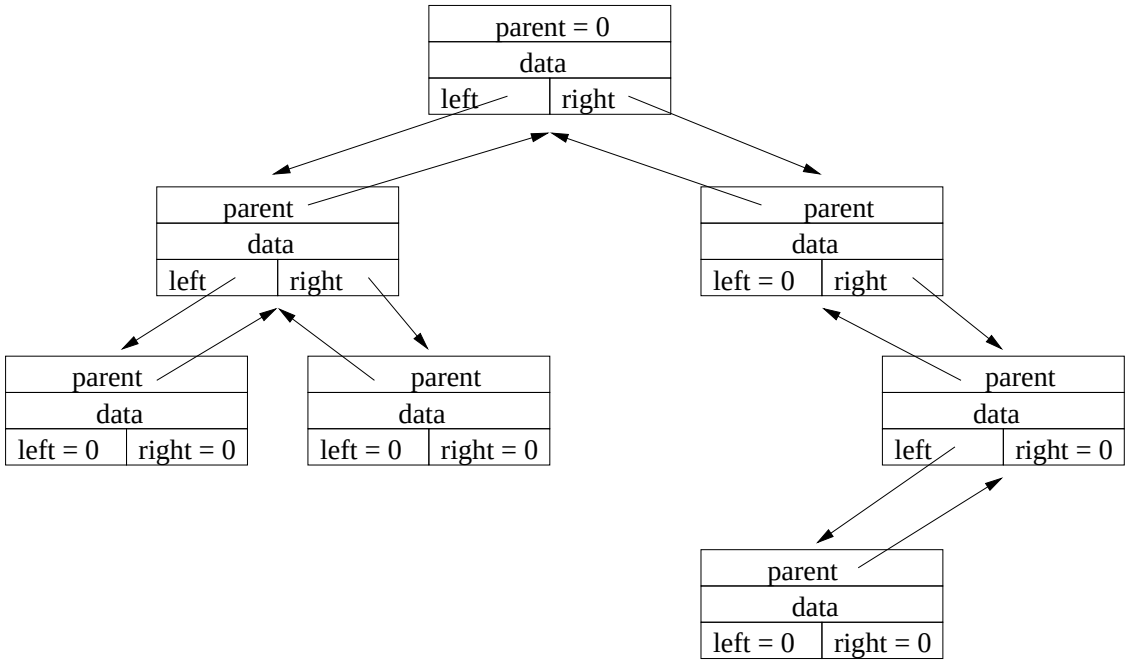


Abbildung 4.2: Verkettung der Knoten eines binären Baumes.

4.7 Traversieren von Bäumen

Erinnern wir uns noch einmal an die abstrakten Datentypen Stack, Queue und Liste: wenn wir eine Operation für alle Elemente ausführen wollten, dann war sofort klar, in welcher Reihenfolge wir die Elemente in dem ADT betrachten wollen/müssen: Bei einem Stack fangen wir mit dem obersten Element an und arbeiten uns nach unten durch, bei einer Queue fangen wir mit dem

ersten Element an und arbeiten uns nach hinten durch und bei einer Liste gehen wir entweder von links nach rechts oder von rechts nach links.

Da die Elemente in Bäumen nicht linear hintereinander sondern über- und untereinander angeordnet sind, muss man bei (Binären) Bäumen einen anderen Weg wählen, um die Elemente zu enumerieren. Wir beschränken uns auch hier wieder auf die Binären Bäume, die Techniken für allgemeine Bäume sind ähnlich.

Die allgemeine (rekursive) Vorgehensweise zur Enumeration der Elemente eines Binären Baumes ist wie folgt:

Algorithmus 4.1 (Enumeration der Elemente eines Baumes) *Angenommen* `R` *ist die Wurzel des zu enumerierenden Baumes.*

- *Definiere eine Funktion*

  `bearbeite_knoten` (K) {
  bearbeite K
  rufe `bearbeite_knoten`(K.left_child)
  rufe `bearbeite_knoten`(K.right_child)
  }

- *Rufe `bearbeite_knoten(R)`.*

Dabei gibt es für die Funktion `bearbeite_knoten` drei Hauptvarianten, die sich in der Reihenfolge unterscheiden, in der K und seine Kinder in der Funktion bearbeitet werden. Wird wie in Algorithmus 4.1 zuerst der Knoten K und dann seine Kinder bearbeitet, so spricht man von **Preorder Traversal**. Bearbeitet man in der Funktion zuerst das linke Kind, dann den Knoten K und dann das rechte Kind (oder umgekehrt), so heißt das **Inorder Traversal**. Kümmert man sich schließlich zuerst um die Kinder und bearbeitet erst dann Knoten K, dann ist das **Postorder Traversal**. Zur Verdeutlichung wollen wir für jede dieser drei Arten ein Beispiel besprechen. Außerdem erläutern wir noch andere Arten, wie Bäume traversiert werden können.

4.7.1 Preorder Traversal

Gegeben sei der Stammbaum aus Abbildung 4.3. Dieser Stammbaum stellt offensichtlich einen Binären Baum dar (auch wenn er ,,auf dem Kopf" steht). Wir möchten nun die verschiedenen Vererbungslinien (ausgehend von ,,Ich") ausgeben, d. h. wir möchten

```
Ich -> Mutter -> Großvater 1
Ich -> Mutter -> Großmutter 1
Ich -> Vater -> Großvater 2
Ich -> Vater -> Großmutter 2
```

ausgeben.

Offensichtlich müssen wir zu diesem Zweck Preorder Traversal anwenden: Zuerst muss ,,Ich" ausgegeben werden, dann ,,Mutter" usw. Außerdem müssen wir noch beachten, dass die internen Knoten ,,Ich", ,,Mutter" und ,,Vater" mehrfach ausgegeben werden – aber darum wollen wir uns hier nicht kümmern.

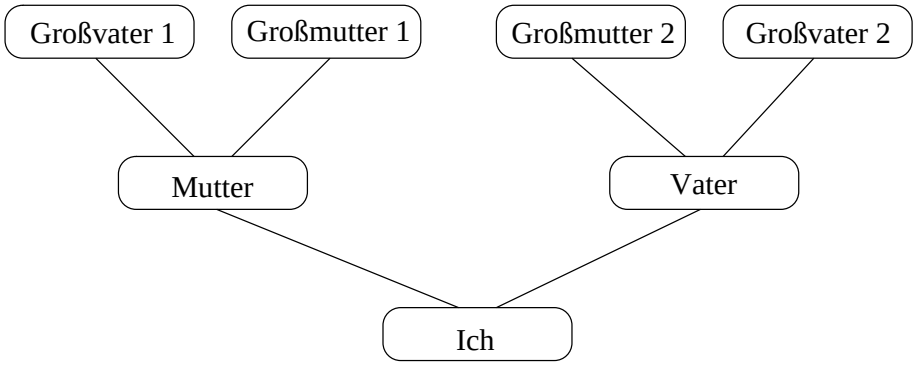


Abbildung 4.3: Binärer Baum als Stammbaum.

4.7.2 Inorder Traversal

Arithmetische Ausdrücke wir $(3 + 4) * 5 - 7$ lassen sich gut als sogenannte **Expression Trees** speichern. Dies sind Binäre Bäume, deren interne Knoten Operatoren und deren externe Knoten Zahlen repräsentieren. Abbildung 4.4 zeigt einen solchen Baum für den Beispielausdruck von oben. Insbesondere sieht man in dieser Abbildung, dass man die Klammern nicht explizit speichern

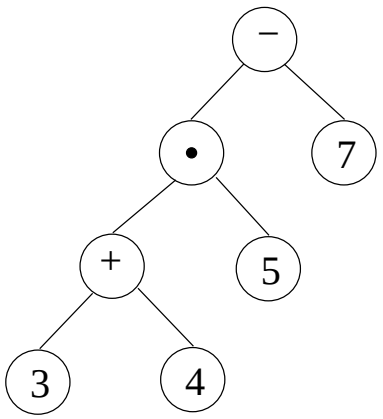


Abbildung 4.4: Expression Tree für $(3 + 4) * 5 - 7$.

muss.

Möchte man den in diesem Baum gespeicherten Ausdruck nun wieder als Text ausgeben, dann muss man Inorder Traversal anwenden: Zuerst muss die 3, dann das +, dann die 4 usw. ausgegeben werden (und nach Bedarf müssen noch Klammern eingefügt werden).

4.7.3 Postorder Traversal

Angenommen wir wollen ein Verzeichnis V mit all seinen Dateien und Unterverzeichnissen löschen. Da man nur leere Verzeichnisse löschen kann, müssen wir zunächst alle Dateien in V löschen, bevor

wir V selbst entfernen können. Enthält V Unterverzeichnisse, dann gilt für diese Entsprechendes: wir müssen erst die Dateien in diesen Unterverzeichnissen löschen, bevor wir die Unterverzeichnisse entfernen können.

Zum Löschen von V betrachten wir den Datei(teil)baum, dessen Wurzel V ist. Die internen Knoten dieses Baumes repräsentieren nichtleere Verzeichnisse, die externen Knoten sind Dateien und leere Verzeichnisse. Sind wir an einem Knoten K angekommen, dann gibt es zwei Möglichkeiten:

1. K hat keine Kinder. In diesem Fall kann K direkt mit einer der Funktionen `remove` (Datei entfernen) oder `rmdir` (leeres Verzeichnis entfernen) gelöscht werden. (In der Shell würden wir entsprechend die Programme `rm` und `rmdir` verwenden.)

2. K hat mindestens ein Kind. In diesem Fall ist K ein nichtleeres Verzeichnis und wir müssen erst all Kinder bearbeiten, d. h. alle Dateien und Unterverzeichnisse in K löschen. Nachdem wir dies getan haben ist K jetzt leer und wir können K mit `rmdir` löschen.

4.7.4 Breitensuche

Betrachtet man sich Preorder Traversal, dann fällt auf, dass man zunächst ganz im Baum hinabsteigt und als erstes den Knoten ,,unten links" bearbeitet. Aus diesem Grund bezeichnet man diese Art von Traversal auch als **Tiefensuche** (*engl.* ,,Depth First Search"). Dabei kommt der Name ,,Suche" daher, dass man in der Praxis häufig deswegen Bäume traversiert, weil man ein Element mit einer bestimmten Eigenschaft finden will (z. B. das größte Element). Auch die Strategien – Inorder und Postorder Traversal – steigen im Baum hinab und bearbeiten den Baum teilbaumweise. Allerdings gibt es auch Situationen, in denen man den Baum Level für Level abarbeiten will, d.h. zuerst die Wurzel, dann alle Knoten auf Level 1, dann alle Knoten auf Level 2 usw. (oder in umgekehrter Reihenfolge). Da man sich bei dieser Strategie erst in die Breite und dann in die Tiefe bewegt, nennt man sie **Breitensuche** (*engl.* ,,Breadth First Search"). Ab und zu spricht man bei dieser Strategie auch von **Level Order Traversal**.

Betrachten wir uns zum Beispiel die Hierarchie in Abbildung 4.5. Hier sind die Offiziere (außer dem Captain) jeweils einem anderen Offizier untergeordnet. Außerdem nimmt die Erfahrung der Offiziere auf einem Level von links nach rechts ab. Angenommen Captain Picard kann das Kommando nicht weiter führen, dann muss der nächstniedrigere, erfahrenste Offizier (Commander Riker) das Kommando übernehmen. Fällt dieser auch aus, ist Commander Data an der Reihe usw. Die Liste der Verantwortlichkeit sieht demnach so aus:

```
1. Captain Picard
2. Commander Riker
3. Commander Data
4. Lt. Cmdr. Worf
5. Lt. Cmdr. LaForge
6. Lt. Cmdr. Crusher
7. Lieutenant Cameo-Appearance
8. Lieutenant Selar
```

Um diese Liste zu generieren, brauchen wir Breitensuche: wir müssen die einzelnen Level nacheinander ausdrucken und auf jedem Level von links nach rechts vorgehen (die Definition von Breitensuche würde auch ,,von rechts nach links" zulassen).

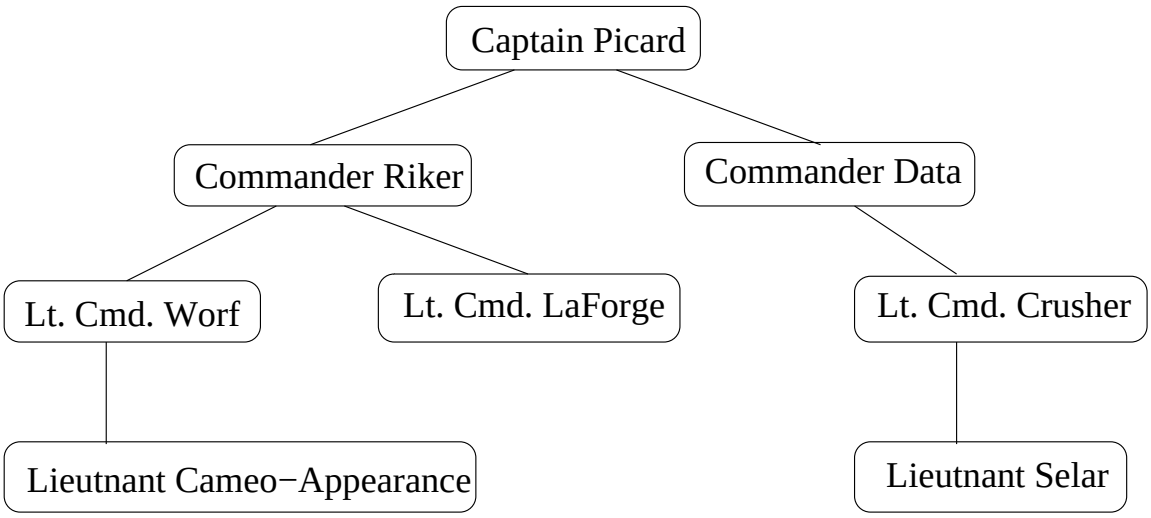


Abbildung 4.5: Ein Baum zur Darstellung einer Hierarchie.

Leider läßt sich die Breitensuche nicht so leicht implementieren wie Pre-, In- und Postorder Traversal. Stattdessen muss man folgenden Algorithmus verwenden:

Algorithmus 4.2 (Breitensuche in einem Baum) *(Der Algorithmus benötigt zusätzlich zu dem eingegeben Baum eine Queue.)*

1. *Initialisiere eine Queue* `Q` *mit der Wurzel des Baumes*
2. *While* `Q` *ist nicht leer*
   (a) `K = dequeue(Q)`
   (b) *Falls* `K` *Kinder hat füge diese am Ende der Queue ein.*
   (c) *Bearbeite* `K`*.*
3. *End While*

Wir brauchen also noch eine Helfer-Datenstruktur, nämlich eine Queue. Man kann sich leicht überlegen, das Algorithmus 4.2 den Baum tatsächlich Level für Level bearbeitet. Ob man auf einem Level von links nach rechts oder rechts nach links geht hängt davon ab, in welcher Reihenfolge man in Schritt 2b die Kinder eines Knotens in die Queue einfügt.

4.7.5 Euler Tour Traversal

Die bisherigen Methoden, über einen Binären Baum zu laufen, hatten eines gemeinsam: jeder Knoten wurde nur *einmal* bearbeitet (besucht). Beim **Euler Tour Traversal** wird dagegen jeder Knoten *dreimal* besucht: einmal von links, einmal von unten, einmal von rechts (siehe Abbildung 4.6). Dabei wird aus Abbildung 4.6 direkt klar, was ,,von links, von unten und von rechts" für interne Knoten heißt. Bei externen Knoten ist das nicht direkt offensichtlich, aber eigentlich auch ganz einfach: ein externer Knoten wird einfach dreimal direkt hintereinander besucht (die Tour ,,geht um den Knoten herum").

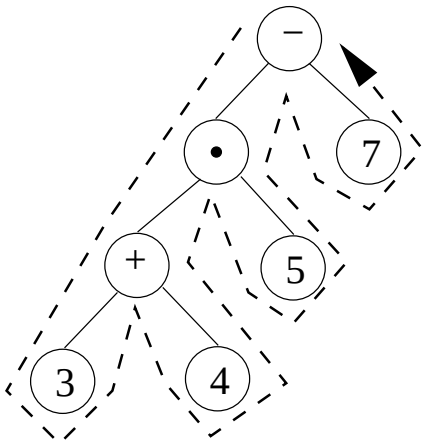


Abbildung 4.6: Euler Tour Traversal in einem Binären Baum.

4.7.6 Verallgemeinertes Inorder Traversal

Die Definitionen von Pre- und Postorder Traversal lassen sich direkt von Binären auf allgemeine Bäume übertragen. Beim Inorder Traversal stellt sich allerdings die Frage: hat ein Knoten mehr als zwei Kinder, zwischen welchen Kindern soll er dann betrachtet werden. Auf diese Frage gibt es grundsätzlich zwei Antworten

1. Je nach Anwendung legt man einfach (willkürlich) eine Stelle zwischen zwei Kindern fest, an der der Knoten bearbeitet werden soll.

2. Man bearbeitet den Knoten *jedesmal*, wenn man einen Kindknoten bearbeitet hat, der nicht der Kindknoten ganz rechts ist.

Die zweite Möglichkeit bezeichnet man als **Verallgemeinertes Inorder Traversal** (*engl.* „Generalized Inorder Traversal"). Wir erwähnen diese Technik hier nur der Vollständigkeit halber und wollen nicht näher darauf eingehen.

Übung: .....

1. Was passiert, wenn man in BFS statt einer Queue einen Stack verwendet (es kommt preoder raus, allerdings von rechts nach links).

2. Euler Tour Traversal implementieren?

3. Inwiefern sind Pre-, In-, PostOrder Traversal Spezialfälle von Euler Tour Traversal.

4. Methoden implementieren

```
/* Process tree node NODE (with optional data DATA).
   Return non-zero if traversal should be stopped and zero otherwise.
*/
typedef int (*process_function)(struct bin_tree_node* node,void* data);
```

```
tree_traverse_preorder (struct bin_tree_node* root,
                        process_function process,
                        void* data); tree_traverse_postorder (struct
bin_tree_node* root,
                        process_function process,
                        void* data); tree_traverse_inorder (struct
bin_tree_node* root,
                        process_function process,
                        void* data); tree_traverse_bfs (struct
bin_tree_node* root,
                        process_function process,
                        void* data);
```

5. Binären Baum speichern und wieder einlesen (mittels Traversals).
6. Binären Baum mit einem Array implementieren.
7. Wieviele Level hat ein „Full Binary Tree" mit n Knoten.

.....

Anhang A

Mehr C-Details

In diesem Anhang gehen wir noch einmal etwas detaillierter auf einige Konzepte in C ein, die nur in den Übungen besprochen wurden. Die meisten der hier aufgelisteten Informationen ließen sich theoretisch auch aus den Manpages zu den einzelnen Funktionen herauslesen. Allerdings hat sich herausgestellt, dass diese Manpages für Anfänger meist schwer zu finden und zu lesen sind.

Nichtsdestotrotz empfehlen wir dringend die Lektüre der einzelnen Man- und Info-Seiten, um sich weiter mit den Konzepten vertraut zu machen.

A.1 Formatierte Ein- und Ausgabe

Wir sind bereits mehrfach auf Anweisungen der Form

```
printf("%d", i);
printf("Die Zahl ist %d.", i);
scanf("%d", &i);
```

gestoßen bzw. haben sie selbst verwendet. Wir wollen nun ein bischen genauer auf die beiden Funktionen `printf()` und `scanf()` eingehen. Im Vergleich zu den Funktionen aus Kapitel 2 haben diese beiden Funktionen eine bisher unbekannte Signatur: sie sind in der Datei `stdio.h` deklariert als

```
int printf (char const* format, ...);
int scanf (char const* format, ...);
```

wobei wir das Schlüsselwort `const` hier nicht weiter beachten.

Diese Deklaration bedeutet, das beiden Funktionen als erstes Argument ein Zeiger auf einen `char` übergeben werden muss. Dieser Zeiger wird als Anfang eines Strings – des sogenannten Format-Strings – betrachtet. Die drei Punkte danach bedeuten, dass nach dem Format-String noch beliebig viele (möglicherweise auch keine) weitere Argumente übergeben werden können. Die Anzahl der übergebenen Elemente wird durch den Format-String festgelegt.

Da sich `printf` und `scanf` in Details unterscheiden, wollen wir uns die beiden Funktionen jeweils einzeln betrachten.

A.1.1 printf

Betrachten wir uns die Funktionsweise von `printf` an folgendem Beispiel:

```
int i = 1;
printf("i hat den Wert %d (was auch sonst?)", i);
```

Der Format-String ist offensichtlich `"i hat den Wert %d (was auch sonst?)"`. Dieser String ist es, der die Ausführung von `printf` steuert. Und zwar wie folgt:

- Der String wird Buchstabe für Buchstabe abgearbeitet.
- Ist der Buchstabe kein Prozentzeichen, dann wird er direkt auf dem Bildschirm ausgegeben.
- Ist der Buchstabe dagegen ein Prozentzeichen, dann wird der nächste Buchstabe (in diesem Fall ein 'd') gelesen. Zu diesem Buchstaben wird das nächste Element aus der Argumentliste (in diesem Fall der Wert der Variablen `i`) herausgesucht. Das Element wird dann in der durch 'd' festgelegten Formatierung (Ausgabe als Dezimalzahl) auf dem Bildschirm ausgegeben.

Neben dem Formatierungsbuchstaben 'd' haben wir bereits die Buchstaben 'c' für die Ausgabe für Buchstaben oder den Buchstaben 's' für die Ausgabe von Strings kennengelernt.

Allgemein kann der Format-String aber wesentlich komplizierter aufgebaut sein (siehe auch Abbildung A.1): nach dem Prozentzeichen muss nicht direkt der Formatierungsbuchstabe (die

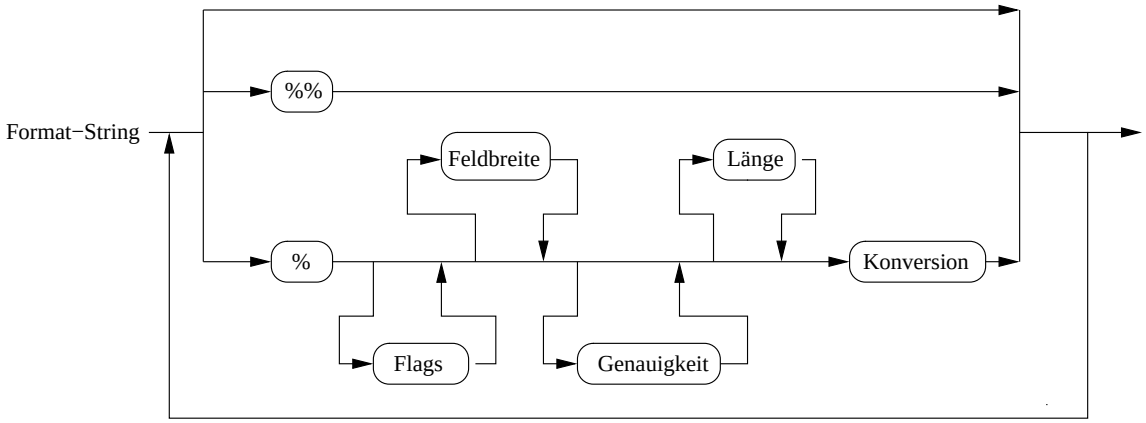


Abbildung A.1: Aufbau eines Format-Strings.

„Konversion") folgen. Stattdessen können jeweils optional Flags, eine Feldbreite, eine Genauigkeit und/oder eine Länge (in dieser Reihenfolge) angegeben werden. Möchte man ein Prozentzeichen ausgeben, so muss man „`%%`" in den Format-String schreiben. Ein Beispiel für solch ein „verbessertes" Format ist

```
int i = 1;
printf("i hat den Wert %+020d (was auch sonst?)", i);
```

Dieses Format sorgt für die folgende Ausgabe:

```
i hat den Wert +00000000000000000001 (was auch sonst?)
```

Wir erläutern nun etwas genauer die einzelnen Bestandteile eines Formats.

Konversion

Die ,,Konversion" besteht aus einem Buchstaben. Sie gibt an, von welchem Typ die auszugebende Variable bzw. der auszugebende Ausdruck ist. Dabei ist im wesentlichen zwischen drei Klassen von Typen zu unterscheiden: ganzen Zahlen, Gleikommazahlen und Buchstaben/Strings.

Ganze Zahlen Diese Zahlen können in Oktal-, Dezimal und Hexadezimaldarstellung ausgegeben werden. Dabei ist zu beachten, dass eine 2-Komplement-Interpretation nur dann erfolgt, wenn die Zahl in Dezimaldarstellung ausgegeben wird. Wird die Zahl in Oktal- oder Hexadezimaldarstellung ausgegeben, dann wird das übergebene Bitmuster als vorzeichenloser Typ interpretiert. Die Buchstaben zur Konversion von ganzen Zahlen sind:

- **d** Die übergebene Zahl ist vom Typ `int` und wird in Dezimaldarstellung ausgegeben.
- **u** Die Zahl ist vom Typ `unsigned int` und wird dezimal ausgegeben.
- **x,X** Die Zahl ist vom Typ `unsigned int` und wird in Hexadezimaldarstellung ausgegeben. Für die (hexadezimalen) Ziffern mit den Zahlenwerten 10 bis 15 werden die Buchstaben 'a' bis 'f' bzw. 'A' bis 'F' verwendet.
- **o,O** Die auszugebende Zahl ist vom Typ `unsigned int` und wird oktal ausgegeben.
- **p** Das Argument für die Konversion ist ein Zeiger. Es wird die in dem Zeiger gespeicherte Adresse in hexadezimaler Notation mit einem führendem '0x' ausgegeben.

Gleitkommazahlen Bei der Ausgabe von Gleitkommazahlen ist zu beachten, dass diese Zahlen vor der Ausgabe gerundet werden.

- **f,F** Das Argument für die Konversion ist vom Typ `double` und wird als [−]ddd.dddddd ausgegeben. Die Anzahl der Dezimalstellen lässt sich durch die Angabe einer Genauigkeit (siehe unten) verändern.
- **e,E** Die Zahl wird in der Form [−]d.dddddde±dd ausgegeben. D.h. vor dem Dezimalpunkt steht immer genau eine Ziffer, danach folgen im Normalfall 6 Ziffern, deren Anzahl sich aber durch die Angabe einer Genauigkeit (siehe unten) ändern lässt. Verwendet man die Konversion 'E' statt 'e', dann wird auch in der Ausgabe ein großes 'E' verwendet.

Strings/Buchstaben Buchstaben werden anhand der ASCII- bzw. Latin1-Tabelle übersetzt und entsprechend ausgegeben.

- **c** Das Argument für die Konversion ist vom Typ `unsigned char` und wird als einzelner Buchstabe ausgegeben.
- **s** Die Konversion betrifft einen Zeiger vom Typ `char*`, der als String interpretiert und ausgegeben wird.

Neben den erwähnten Konversionen gibt es noch eine Reihe anderer, die wir aber im Normalfall nicht benötigen.

Länge

Für einige Typen wie z.B. `unsigned short` ist keine explizite Konversion definiert. Für die Ausgabe solcher Typen muss die Konversion mit einem Längenmodifikator versehen werden. Diese Modifikatoren sind:

h Die auszugebende ganze Zahl ist vom Typ `short` ('%hd') oder `unsigned short` ('%hu').

l Die auszugebende ganze Zahl ist vom Typ `long` ('%ld') oder `unsigned long` ('%lu').

ll Die auszugebende Zahl ist vom Typ `long long` ('%lld') oder `unsigned long long` ('%llu').

L Die auszugebende Gleitkommazahl ist vom Typ `long double` (z.B. '%Lf').

Auf der Manpage von `printf` finden sich noch weitere Modifikatoren, die aber im Allgemeinen nicht gebraucht werden.

Flags

Flags betreffen aussschließlich die Ausgabe von Zahlen. Die meisten von ihnen sind nur dann relevant, wenn die Zahl mit einer Feldbreite ausgegeben wird (siehe unten).

'0' Wird eine Zahl mit einer Feldbreite ausgegeben und die Darstellung der Zahl benötigt weniger Ziffern als die Feldbreite, dann werden die überschüssigen Ziffern von links mit Nullen aufgefüllt (im englischen nennt man das **zero padding**).

'-' Bei Ausgabe mit einer Feldbreite werden an die Darstellung der Zahl solange Leerzeichen angehängt, bis die Feldbreite erreicht ist.

' ' (Blank) Wenn dieses Flag angegeben ist, wird eine negative Zahl mit einem führenden '-' und eine positive Zahl mit einem führenden Leerzeichen ausgegeben.

'+' Sorgt dafür das die Zahl mit Vorzeichen ('+' oder '-') ausgegeben wird.

Zusätzlich zu diesen Flags gibt es noch einige weitere, die man aber normalerweise nicht braucht und die wir deshalb hier nicht erwähnen.

Feldbreite

Die Feldbreite muss als Dezimalzahl ohne führende Nullen angegeben werden. Genau wie bei den Flags ist die Angabe einer Feldbreite nur dann sinnvoll, wenn Zahlen ausgegeben werden. Sie sorgt dafür, dass Zahlen mindestens mit der angegebenen Anzahl an Zeichen dargestellt werden. Werden für die Darstellung der Zahl weniger Zeichen benötigt, dann wird von links mit Leerzeichen bis zur Feldbreite aufgefüllt.

Genauigkeit

Die Genauigkeit gibt an, mit wievielen Nachkommastellen eine Gleitkommazahl ausgegeben wird. Demnach ist die Angabe einer Genauigkeit nur dann sinnvoll, wenn Variablen oder Ausdrücke vom typ `float` oder `double` (oder `long double`) ausgegeben werden.

Die Angabe der Genauigkeit besteht aus einem Dezimalpunkt gefolgt von einer Zahl, die die Anzahl der Nachkommastellen angibt.

A.1.2 scanf

Relativ analog zu `printf` funktioniert `scanf`. Die einzigen Unterschiede hier sind:

1. Man gibt nicht an, mit welchem Format ausgegeben wird, sondern mit welchem Format eingelesen wird.

2. Man muss in der Argumentliste jeweils nicht die Variablen oder Ausdrücke angegeben, sondern Adressen von Variablen.

Am einfachsten lässt sich die Funktionsweise von `scanf` wieder an einem Beispiel demonstrieren:

```
int i = -1;
...
scanf("Die Zahl soll %d sein.", &i);
printf("Die Zahl war %d", i);
```

Dieses Codefragment erwartet, dass der Benutzer die Zeichenkette

```
Die Zahl soll xxx sein.
```

eingibt, wobei `xxx` durch die Dezimaldarstellung einer Zahl ersetzt wird. `xxx` wird dann in `i` gespeichert. Gibt der Benutzer etwas anderes als die geforderte Zeichenkette ein, dann wird der Wert von `i` nicht verändert.

Allerdings gibt es hierbei eine wichtige Ausnahme: Ein Leerzeichen im Format-String steht für eine *beliebige nichtleere* Menge von Whitespaces in der Eingabe. Im obigen Fall kann der Benutzer statt einem Leerzeichen zwischen zwei Worten also auch mehrere Leerzeichen, Tabulatoren und sogar Zeilenumbrüche verwenden!

A.1.3 Rückgabewerte

Wie aus der Definition von `printf` und `scanf` ersichtlich ist, geben beide Funktionen einen Wert vom Typ `int` zurück. Bei `printf` ist dies die Anzahl der geschriebenen Buchstaben, bei `scanf` die Anzahl der erfolgreich bearbeiteten Formate.

A.1.4 Fehler im Format

Die meisten modernen Compiler können bereits während dem Übersetzen eines Programms feststellen, ob die Argumente in der Liste zu den angegebenen Formatierungsbuchstaben passen. Ist dem nicht so, dann wird eine Warnung ausgegeben, die man durchaus ernst nehmen sollte.

Gibt man z.B. für ein „%s"-Format statt einem `char*` einen `int` an, dann wird der Inhalt dieses `int` als Adresse des ersten Buchstaben eines Strings interpretiert. Die Werte, die zufällig an dieser Speicherstelle stehen, werden dann als String ausgegeben. Darf man gar nicht auf die entsprechende Speicherstelle zugreifen, dann stürzt das Programm ab.

A.1.5 Pufferung

Die Ausgabe von `printf` erfolgt **gepuffert** (*engl.* ,,buffered"), d.h. alle Ausgaben werden erst in einen internen Speicher geschrieben. Dieser interne Speicher wird erst dann auf dem Bildschirm ausgegeben, wenn er voll ist oder wenn ein Zeilenvorschub (,,`\n`") geschrieben wird.

Am besten versteht man dies an einem Beispiel:

```
for (i = 0; i < 10; i++) {
   printf(".");
   sleep(1);
}
printf("\n");
```

Man könnte zunächst meinen, das Codefragment gibt jeweils einen Punkt aus, wartet eine Sekunde, gibt dann den nächsten Punkt aus usw.

Wenn man das Programm allerdings laufen lässt, stellt man fest, dass sich zunächst zehn Sekunden lang gar nichts tut und dann alle Punkte auf einmal ausgegeben werden.

Wer daran interessiert ist, wie man diese Pufferung abschaltet, der möge sich die Manpages zu den Funktionen `fflush` und `setbuffer` ansehen.

A.2 Ein- und Ausgabe mit Dateien

Neben den beiden Funktionen

```
int printf(char const* format, ...);
int scanf(char const* format, ...);
```

die zur formatierten Ein- und Ausgabe mit Tastatur und Bildschirm dienen und bereits in Abschnitt A.1 erläutert wurden, gibt es die zwei Funktionen

```
int fprintf(FILE* f, char const* format, ...);
int fscanf(FILE* f, char const* format, ...);
```

Mit diesen beiden Funktionen ist die formatierte Ein- und Ausgabe mit Dateien möglich und funktioniert analog zu `printf` und `scanf`.

Eine Datei wird in `C` durch einen Zeiger `FILE*` repräsentiert. Wie der Datentyp `FILE` dabei genau aussieht hängt von dem verwendeten Betriebssystem ab und soll uns hier nicht weiter interessieren.

Bevor man eine Datei benutzen kann, muss man sie zunächst mit Hilfe der Funktion `fopen()` öffnen. Ist die Arbeit an der Datei beendet, dann muss die Datei mit `fclose()` wieder geschlossen werden.

A.2.1 `fopen()`

Die Funktion `fopen` ist in `stdio.h` deklariert und hat die folgende Signatur (siehe auch `man fopen`):

```
FILE* fopen (char const* path, char const* mode);
```

Dabei bezeichnet `path` den Pfad der Datei, die geöffnet werden soll und `mode` den Modus, in dem die Datei geöffnet wird. Dabei unterscheiden wir die drei folgenden Modi

`"r"` Die Datei wird zum Lesen (read) geöffnet. Existiert die angegebene Datei nicht, so ist dies ein Fehler.

`"w"` Die Datei wird zum Schreiben (write) geöffnet. Existiert die angegebene Datei, dann wird sie überschrieben. Dabei geht der urspüngliche Inhalt der Datei mit bereits dem Öffnen verloren! Existiert die angegebene Datei nicht, dann wird eine entsprechende neue Datei angelegt.

`"a"` Die Datei wird zum ,,Anhängen" (append) geöffnet. Existiert die Datei noch nicht, ist dieser Modus äquivalent zu `"w"`. Gibt es dagegen schon eine entsprechende Datei, dann wird die Datei nicht überschrieben, sondern Schreiboperationen hängen neue Zeichen hinten an die Datei an.

Der von `fopen` zurückgegebene Zeiger kann dann als Argument für `fprintf` oder `fscanf` benutzt werden. Dabei ist zu beachten, dass eine Datei, aus der gelesen werden soll, natürlich auch zum Lesen geöffnet worden sein muss. Entsprechendes gilt für Dateien, in die geschrieben werden soll.

Tritt beim Öffnen der Datei ein Fehler auf, dann wird der **Nullzeiger** (*engl.* ,,null pointer") zurückgegeben. Dies ist ein Zeiger, der die Adresse 0 speichert. Somit lässt sich wie folgt überprüfen, ob die Datei tatsächlich geöffnet werden konnte:

```
FILE* f;
...
f = fopen("Datei", "r");
if (! f) {
   printf("Die Datei %s konnte nicht zum Lesen geöffnet werden!\n ",
"Datei");
}
```

A.2.2 `fclose`

Eine Datei, die mit `fopen)` geöffnet wurde, muss genau einmal mit

```
int fclose(FILE* f);
```

wieder geschlossen werden. Dabei sorgt die Funktion `fclose` insbesondere dafür, dass eventuell gepufferte und noch nicht ausgegebene Daten (siehe Abschnitt A.1.5) in die Datei geschrieben werden, bevor die Datei geschlossen wird. Anderenfalls würden diese Daten verlorengehen.

A.2.3 Standard Ein- und Ausgabe

In jedem C-Programm stehen immer die drei folgenden ,,Dateien" zur Verfügung:

```
FILE* stdin; /* standard input*/
FILE* stdout; /* standard output */
FILE* stderr; /* standard error */
```

Diese drei Dateien werden beim Start des Programms automatisch geöffnet und am Ende des Programms automatisch wieder geschlossen.

Dabei werden `stdin` zum Lesen und `stdout` und `stderr` zum Schreiben geöffnet.

Der Standard-Input eines Prozesses ist normalerweise die Tastatur. Deswegen entspricht das Lesen von `stdin` dem Lesen von der Tastatur und `fscanf(stdin, ...)` ist das gleiche wie `scanf(...)`.

Der Standard-Output eines Prozesses ist in Normalfall die Konsole. Ein Schreiben auf `stdout` bewirkt deswegen eine Ausgabe auf dem Bildschirm und `fprintf(stdout, ...)` und `printf(...)` sind das gleiche.

Der Standard-Error-Output eines Prozesses ist ein Konzept, das wir bisher noch nicht kennengelernt haben: er wird dazu verwendet, Fehlermeldungen des Prozesses auszugeben. Normalerweise werden Zeichen, die auf `stderr` geschrieben werden, auch auf dem Bildschirm ausgegeben. Es existieren aber auch Möglichkeiten, die Ausgaben auf `stderr` oder `stdout` zu unterdrücken. So kann man dafür sorgen, dass nur noch die Fehlermeldungen oder eben gar keine Fehlermeldungen mehr auf dem Bildschirm ausgegeben werden. Obwohl wir uns in dieser Veranstaltung nicht weiter mit diesem Konzept beschäftigen werden, geben wir Fehlermeldungen immer wie folgt aus:

```
if ( /* irgendwas stimmt nicht */) {
   fprintf(stderr, "Irgendwas ist nicht so, wie es sein soll!");
}
```

A.3 Weitere Schlüsselworte

Wie wir bereits in Tabelle 2.1 in Kapitel 2 gesehen haben, gibt es in `C` mehr Schlüsselwörter als wir im Hauptteil dieses Dokuments vorgestellt haben. In diesem Abschnitt wollen wir nun noch einige weitere Schlüsselwörter erläutern.

A.3.1 `sizeof`

Das Schlüsselwort `sizeof` ist besonders in Verbindung mit den Funktionen `malloc()`, `realloc()` und `free()` wichtig. Der Ausdruck

```
sizeof(x)
```

liefert die Speichergröße von `x` in Bytes zurück. Möchte man z.B. ein Array von 20 `double` allokieren, dann macht man das so:

```
double* array;
...
array = malloc(sizeof(double) * 20);
if (! array) {
   /* out of memory */
   ...
}
```

Wie das Beispiel zeigt, muss man der ,,Funktion" `sizeof` nicht unbedingt eine Variable übergeben, sondern kann auch direkt einen Typ angeben. Der Rückgabewert ist dann die Anzahl an Bytes, die benötigt werden, um eine Variable diesen Typs zu speichern.

A.3.2 typedef

Mit Hilfe von `typedef` kann man – ähnlich wie mit Makros – Abkürzungen definieren. Allerdings sind die Abkürzungen mit `typedef` nicht einfach Abkürzungen, sondern echte Typ-Deklarationen. Die Syntax einer `typedef`-Anweisung ist wie folgt:

```
typedef <alter Name> <neuer Name>
```

Dabei ist *alter Name* der Name eines bereits existierenden Typs (also z.B. `int`) und *neuer Name* ist ein neuer Name für den Typ. Häufig verwendete `typedef`-Anweisungen sind

```
typedef unsigned long ulong;
typedef unsigned short ushort;
```

Diese Zeilen definieren die neuen Typen `ulong` und `ushort`, die einem `unsigned long` bzw. einem `unsigned short` entsprechen.

A.3.3 break und continue

Angenommen, wir wollen ein Programm schreiben, dass in einer Schleife zwei Zahlen a und b ausgibt und dann a Prozent von b berechnet und ausgibt. Dies soll solange wiederholt werden, bis $a = 0$ eingegeben wird. Ausserdem soll a immer größer als Null sein, d.h. wenn $a < 0$ ist, wird erneut nach a gefragt.

Eine erste Implementierung könnte so aussehen:

```
#include <stdio.h>

int main (void)
{
  double a, b;
  do {
    printf("a? "); scanf("%f", &a);
    if (a > =.) {
      printf("b? "); scanf("%f", %b);
      printf("%f percent of %f: %f", a, b, (a / 100) * b);
    } else if (a < 0.) {
      printf("a must be non-negative! Try again!\n");
    }
  } while (a != 0.);

  return 0;
}
```

Diese Lösung ist relativ unübersichtlich und kann mit Hilfe der beiden Schlüsselwörter `break` und `continue` verbessert werden:

```
#include <stdio.h>

int main (void)
```

```
{
  double a, b;
  do {
    printf("a? "); scanf("%f", &a);

    if (a < 0.) {
      printf("a must be non-negative! Try again!\n");
      continue;
    }
    if (a == 0.) break;

    printf("b? "); scanf("%f", %b);
    printf("%f percent of %f: %f", a, b, (a / 100) * b);

  } while (a != 0.);

  return 0;
}
```

Dabei bewirken die beiden neuen Schlüsselwörter folgendes

`break` An dieser Stelle wird die Schleife sofort abgebrochen. Egal ob die Schleifenbedingung „`a != 0`" wahr ist oder nicht, wird mit der ersten Zeile nach der Schleife (in diesem Fall „`return 0`") weitergemacht.

`continue` Es wird an das Ende der Schleife gesprungen. Der Ausdruck „`a != 0`" wird ausgewertet und entsprechend dem Ergebnis wird die Schleife fortgeführt oder beendet (in unserem Fall ist das Ergebnis immer falsch, also wird die Schleife neu gestartet).

Genau wie in dem Beispiel, können `break` und `continue` auch in `for`- und `while`-Schleifen eingesetzt werden. Außerhalb von Schleifen ist (abgesehen von einer kleinen Ausnahme) die Verwendung von `break` und `continue` nicht erlaubt.

A.3.4 `volatile` und `const`

Häufig werden Variablen nur einmal initialisiert und ändern dann ihren Wert nicht mehr. Solche Variablen werden im Maschinencode dann eigentlich nicht mehr als Variable benötigt: überall wo diese Variable auftritt, kann der direkt ihr Wert eingesetzt werden.

Um dem Compiler anzuzeigen, dass eine Variable konstant ist, verwendet man das Schlüsselwort `const`, z.B.:

```
long const max_long = 2147483647;
```

Dabei ist zu beachten, dass `const` in diesem Fall zum Typ dazugehört, d.h. `long` und `long const` sind zwei verschieden Typen. Selbstverständlich kann man einem konstanten Typen keine Werte mehr zuweisen. Eine Anweisung wie

```
max_long = 1;
```

würde vom Compiler nicht akzeptiert.

Neben der Verwendung bei konstanten Variablen ist die Verwendung von `const` auch bei Funktionsargumenten möglich. Wir haben schon einige Funktionen gesehen, die ein Argument vom Typ `char const*` übergeben bekamen:

```
int printf (char const* format, ...);
size_t strlen (char const* s);
char* strdup (char const* s);
```

Allen Funktionen ist eins gemeinsam: Der übergebene String wird nicht verändert. An dieser Stelle ist das eine ,,Garantie", die diese Funktionen an den Aufrufenden geben: in dieser Funktion wird der String nicht verändert. Denn jegliche Änderung des Strings in der Funktion würde wegen des `const` zu einem Fehler bei der Übersetzung führen.

Die Verwendung von `const` hat also zwei Aspekte:

- zum einen teilt sie dem Compiler mit, welche Variablen ihren Wert nicht mehr ändern werden und deshalb ,,wegoptimiert" werden können,
- zum anderen stellt sie sicher, dass bestimmte Variablen nicht mehr geändert werden können – auch nicht aus Versehen.

Eine korrekte und konsistente Verwendung von `const` hilft einem, viele Fehler und Bugs zu vermeiden.

Leider ist nicht eindeutig festgelegt, an welcher Stelle des Typs das `const` stehen muss. So bezeichnen z.B.

`long const` und `const long`

den gleichen Typ. Der Übersicht halber einigen wir für diese Veranstaltung darauf, dass wir das `const` immer soweit wie möglich nach rechts schreiben, sodass alles links vom `const` konstant ist. Das ist insbesondere dann von Interesse, wenn man sich Deklarationen wie

```
char const* const* text = "Ein bischen Text"
```

betrachtet, in denen mehrere `const` auf einmal auftauchen (Details hierzu werden in den Übungen behandelt).

Das Schlüsselwort `volatile` bezeichnet in einem gewissen Sinne das Gegenteil von `const` – allerdings auch nicht immer. Da `volatile` aber nur interessant ist, wenn man mit mehreren Threads arbeitet und wir dies nicht tun werden, gehen wir hier nicht weiter darauf ein.

A.3.5 `switch`, `case` und `default` (und `break`)

Angenommen wir schreiben ein Programm `calculate`, das einen arithmetischen Ausdruck berechnet, der auf der Kommandozeile übergeben wird. Die Eingabe von

```
fb04305: ~> ./calculate 4 + 5
```

soll also zur Ausgabe von ,,9" auf dem Bildschirm führen.

Ausser dem Operator '+' soll das Programm noch die Operatoren '-', '*' und '/' interpretieren können. Ein erster Ansatz des Programms könnte dann wie folgt aussehen

```
int main (int argc, char* argv[]) {    int a = strtol(argv[1], 0, 10);
   int b = strtol(argv[3], 0, 10);
   int res = 0;

   if (argv[2][0] == '+') res = a + b;
   else if (argv[2][0] == '-') res = a - b;
   else if (argv[2][0] == '*') res = a * b;
   else if (argv[2][0] == '/') res = a / b;
   else printf("Unknown operator %c!", argv[2][0]);
   printf("%d\n", res);
   return 0;
}
```

Dabei ist die Folge von ,,`if`"- und ,,`else if`" sehr lang und unübersichtlich. Unter Verwendung der Schlüsselwörter `switch`, `case`, `default` und `break` lässt sich der Code kompakter schreiben:

```
int main (int argc, char* argv[]) {    int a = strtol(argv[1], 0, 10);
   int b = strtol(argv[3], 0, 10);
   int res = 0;

   switch (argv[2][0]) {
      case '+': res = a + b; break;
      case '-': res = a - b; break;
      case '*': res = a * b; break;
      case '/': res = a / b; break;
      default: printf("Unknown operator %c!", argv[2][0]);
   }
   printf("%d\n", res);
   return 0;
}
```

Dabei funktioniert die `switch`-Anweisung wie folgt:

- Der in Klammern angegebene Ausdruck (in diesem Fall ,,`argv[2][0]`") wird ausgewertet und liefert den Wert d.
- Unter den verschiedenen angegebenen Fällen (,,`case`") wird der *erste* herausgesucht, der auf d passt. Dabei passt `default` auf alle Werte von d und ansonsten ,,`case e`" auf d genau dann, wenn $d = e$ gilt.
- Alle Anweisungen, die dem Doppelpunkt folgen werden ausgeführt. dabei werden die Anweisungen so lange durchgeführt, bis das Ende der `switch`-Anweisung erreicht ist oder die nächste Anweisung ein `break` ist. Im letzteren Fall wird zum Ende der `switch`-Anweisung gesprungen.

Die Beschreibung von `switch` impliziert einige Besonderheiten, die auf den ersten Blick nicht offensichtlich sind:

Anweisungsliste

Hinter einem ,,`case ...:`" oder einem ,,`default:`" kann eine beliebig lange Liste von Anweisungen stehen (auch eine leere Liste ist erlaubt). Allerdings wird diese Liste *nicht* als Block betrachtet, d.h. an ihrem Anfang können keine neuen Variablen definiert werden. Möchte man trotzdem neue Variablen in dieser Liste definieren, dann muss man sich mit einem Block behelfen:

```
case ...:
   {
      int i;
      ...
   }
break;
```

Allerdings sollte man immer auch darauf achten, dass Block und Anweisungsliste nicht zu groß werden, da ansonsten die Übersichtlichkeit leidet. Im Notfall muss man die Anweisungsliste eben in entsprechende Funktionen auslagern.

Fallthrough

Betrachten wir uns das folgende Code-Fragment

```
int len = 0;
int as = 0;
char* s = ...;
for (i = 0; s[i]; i++) {
   switch (s[i]) {
      case 'a': as++;
      default: len++;
   } }
```

Hierbei ist insbesondere zu beachten, dass hinter dem `as++` *mit Absicht kein* `break` steht. Das bewirkt Folgendes:

- Ist der aktuelle Buchstabe kein 'a', dann wird das `default`-Label angesprungen und `len` um 1 erhöht.
- Ist der aktuelle Buchstabe dagegegen ein 'a', dann wird zunächst das 'a'-Label angesprungen und `as` um 1 erhöht. Nach der Definition von oben wird nun mit der Ausführung der nächsten Anweisung fortgefahren, bis ein `break` oder das Ende der `switch`-Anweisung erreicht werden. Die nächste ,,Anweisung" ist ,,`default`": sie bewirkt gar nichts. Die folgende Anweisung ist `len++` und `len` wird dementsprechend um 1 erhöht. Nun ist das Ende der `switch`-Anweisung erreicht und es wird wieder zum Kopf der Schleife gesprungen.

Damit ist klar, dass das Code-Fragment in `len` die Anzahl Buchstaben in `s` und in `as` die Anzahl kleiner as in `s` zählt.

Das ,,Durchfallen" bei fehlendem `break` von einem Fall in den darunter bezeichnet man als **fallthrough**.

Mehrere Fälle gemeinsam abhandeln

Die Anweisungsliste hinter einem „`case ...:`" darf auch leer sein. So lassen sich mehrere Fälle bündeln:

```
int vowels = 0;
int consonants = 0;
char* s = ...;
for (i = 0; s[i]; i++) {
   switch (s[i]) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u': vowels++; break;
      default: consonants++;
   }
}
```

Dieses Code-Fragment zählt in `vowels` die Anzahl Vokale und in `consonants` die Anzahl Konsonanten in `s`.

A.3.6 `enum`

Ähnlich wie mit Makros und dem Schlüsselwort `const` kann man auch mit dem Schlüsselwort `enum` Konstanten definieren. Allerdings kann man mit `enum` nur ganzzahlige Konstanten einer bestimmten Art definieren.

Mit

```
enum { Konstante_1, Konstante_2, Konstante_3 };
```

werden drei ganzzahlige Konstanten `Konstante_1`, `Konstante_2` und `Konstante_3` mit den Werten 0, 1 und 2 definiert. Das erste Element in so einer Definition repräsentiert den Wert 0, das zweite den Wert 1 usw. Anders als Variablen, die als `const` deklariert sind, werden `enum`-Elemente beim Compilieren in jedem Fall durch die entsprechenden Werte ersetzt.

Beliebige Werte in `enum`

Man ist nicht gezwungen, für die Werte in der `enum` aufeinanderfolgende Werte zu verwenden, die bei Null beginnen. Stattdessen kann man jedem Element der `enum` einen Wert explizit zuweisen. Beliebt sind z.B. Konstrukte der Art

```
enum { bit0 = 1, bit1 = 2, bit2 = 4, ..., bit8 = 128 };
```

Ebenso ist es erlaubt, in der `enum` mehrere Elemente mit dem gleichen Wert zu definieren:

```
enum { bit0 = 1, bitright = 1, bit1 = 2, bit3 = 4, ...,
       bit8 = 128, bitleft = bit8 };
```

Zu guter letzt kann man auch noch negative Zahlen zuweisen oder alle Zuweisungstypen vermischen:

```
enum {
      artificial1, artificial2, artificial3 = -1, artificial4,
      artificial5 = artificial2, artificial6
}
```

Dieses Beispiel definiert die folgenden Elemente mit den angegebenen Werten:

artificial1	0
artificial2	1
artificial3	-1
artificial4	0
artificial5	1
artificial6	2

Damit ist klar: gibt man für ein Element der `enum` keinen expliziten Wert an, dann ist der Wert des Elements immer der um eins erhöhte Wert des vorigen Elements.

`enum` als Typ

Eine Enumeration, die mit `enum` definiert wird, lässt sich auch als eigener Typ definieren, und zwar auf die zwei folgenden verschiedenen Arten:

```
enum enumeration_1 { Wert_1_1, Wert_1_2, ... };
typedef enum { Wert_2_1, Wert_2_2, ... } enumeration_2;
```

Die erste Variante definiert einen neuen Typ „`enum enumeration_1`" (hier gehört `enum` zum Typnamen!), die zweite einen Typ „`enumeration_2`" (hier gehört `enum` *nicht* zum Typnamen!). Beide Typen können nach dieser Definition ganz normal wie Typen verwendet werden.

Allerdings kann man Variablen diesen Typs immer noch jede beliebige ganze Zahl zuweisen, denn `gcc` überprüft leider nicht, ob dieser Wert auch in der entsprechenden `enum` vorhanden ist.

`enum` und `switch`

Besonders gut eignet sich `enum` zur Verwendung mit `switch`. Angenommen, man hat ein Programm, dass 3 verschiedene Zustände annehmen kann. Der aktuelle Zustand soll in einer Variablen `state` gespeichert sein. Von Zeit zu Zeit muss in Abhängigkeit vom aktuellen Zustand eine bestimmte Aktion durchgeführt werden. Das könnte z.B. so aus

```
typedef enum { state_1, state_2, state_3 } state_t;
...
state_t state = ...;
...
switch (state) {
   case state_1: ...; break;
   case state_3: ...; break;
}
```

Im Gegensatz zu als `const` deklarierten Variablen, können `enum`-Werte also auch als `case`-Label in `switch`-Statements verwendet werden.

Bei der Verwendung eines `enum`-Wertes in einem `switch`-Statement ergibt sich außerdem noch folgender Vorteil: Wie man leicht sieht, existiert in obigem Code-Fragment kein `case`-Label für den Fall `state == state_2`. Der Compiler kann dies erkennen und gibt eine entsprechende Warnung aus:

```
warning: enumeration value ‘state_2’ not handled in switch
```

Dies hilft dem Programmierer, keine Fälle zu vergessen.

Dabei ist zu beachten, dass natürlich keine Warnung ausgegeben wird, wenn für einen Wert der `enum` kein `case`-Label definiert ist, stattdessen aber ein `default`-Label existiert.

A.3.7 `goto` und `union`

Die beiden Schlüsselwörter `goto` und `union` dienen völlig verschiedenen Zwecken. Allerdings werden beide nur in äusserst seltenen Fällen benötigt. Solange man nicht den Beruf eines Systemprogrammierers ergreift wird man im Normalfall keines dieser beiden jemals benutzen müssen. Deshalb gehen wir hier nicht weiter auf die beiden Schlüsselwörter ein.

A.3.8 `auto` und `register`

Die Schlüsselwörter `auto` und `register` bezeichnen die *Speicherklasse* einer Variablen. Sie sind im Grunde nur noch ein Relikt vergangener Tage, in denen die `C`-Compiler noch nicht so leistungsfähig waren. Bei der Leistungsfähigkeit der heutigen Compiler erübrigt sich für uns die Anwendung dieser Schlüsselwörter.

A.4 Zeigerarithmetik

In `C` sind arithmetische Operationen nicht nur zwischen Zahlentypen, sondern auch zwischen Zeigertypen erlaubt. So kann man z.B. zwei Zeiger addieren oder subtrahieren oder zu einem Zeiger eine konstante Zahl addieren.

In diesen Operationen wird der ,,Wert" eines Zeigers durch seine Adresse bestimmt, d.h. die Summe zweier Zeiger speichert die Summe der beiden Adressen.

Allerdings gibt es hierbei eine wichtige Ausnahme:

> Ist T ein Datentyp, P ein Zeiger auf T, b die Anzahl Bytes die eine Variable vom Typ T belegt und k eine ganze Zahl (die auch negativ sein kann). Dann ist
>
> $$\text{addr}(P + k) = \text{addr}(P) + k \cdot b,$$
>
> wobei $\text{addr}(P)$ die in P gespeicherte Adresse bezeichnet.

Diese Besonderheit lässt sich am einfachsten an einem Beispiel erklären:

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

```
int main (void)
{
  char char_array[10] = { '0', '1', '2', '3', '4',
  '5', '6', '7', '8', '9' };
  long long_array[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  char* c = char_array;
  long* l = long_array;
  int i;

  for (i = 0; i < 10; i++) {
    printf("char: %p -> %c, long %p -> %ld\n", c, *c, l, *l);
    c = c + 1;
    l = l + 1;
  }

  return 0;
}
```

Dieses Programm druckt

```
char: 0xbffff800 -> 0, long 0xbffff7d8 -> 0
char: 0xbffff801 -> 1, long 0xbffff7dc -> 1
char: 0xbffff802 -> 2, long 0xbffff7e0 -> 2
char: 0xbffff803 -> 3, long 0xbffff7e4 -> 3
char: 0xbffff804 -> 4, long 0xbffff7e8 -> 4
char: 0xbffff805 -> 5, long 0xbffff7ec -> 5
char: 0xbffff806 -> 6, long 0xbffff7f0 -> 6
char: 0xbffff807 -> 7, long 0xbffff7f4 -> 7
char: 0xbffff808 -> 8, long 0xbffff7f8 -> 8
char: 0xbffff809 -> 9, long 0xbffff7fc -> 9
```

auf dem Bildschirm. Warum?

Nach der Initialisierung der Variablen ergibt sich im Speicher das Bild in Abbildung A.2:

- An Adresse 0xbffff800 wurde das 10 `char` lange Array `char_array` angelegt. Da ein `char` genau ein Byte belegt, umfasst dieses Array 10 Byte.
- An Adresse 0xbffff7d8 wurde das 10 `long` lange Array `long_array` angelegt. Da ein `long` genau ein Byte belegt, umfasst dieses Array 40 Byte.
- Die Variable `c` zeigt auf `char_array` bzw. dessen erstes Element und die Variable `l` zeigt auf `long_array` bzw. dessen erstes Element.

Damit sollte die Ausgabe der ersten `printf()`-Anweisung klar sein.

Nach der ersten Ausgabe werden sowohl `c` als auch `l` um eins erhöht. Da `c` vom Typ `char*` ist, bedeutet das, dass zu der in `c` gespeicherten Adresse die Größe eines `char` in Byte addiert wird. Die in `c` gespeicherte Adresse wird also um 1 erhöht und `c` zeigt jetzt auf das zweite Element von `char_array` (siehe auch Abbildung A.3).

Da ein `long` im Gegensatz zu einem `char` allerdings 4 Byte umfasst, bewirkt die Anweisung `l = l + 1`, dass zu der in `l` gespeicherten Adresse 4 Byte addiert werden und `l` somit auf das zweite Element von `long_array` zeigt (siehe auch Abbildung A.3).

Auf diese Art wird nun über alle Elemente der beiden Arrays `char_array` und `long_array` iteriert, wobei die Additionen

```
c = c + 1;
l = l + 1;
```

jeweils automatisch das Richtige tun.

Ähnlich wie die Addition einer Konstanten zu einem Zeiger wird auch die Differenz von zwei Zeigern interpretiert:

> Die Differenz zwischen zwei Zeigern P_1 und P_2 auf den Typ T ist die Anzahl der Elemente vom Typ T, die noch zwischen die beiden Zeiger gepackt werden können (bzw. das Inverse dieser Zahl).

Wenn wir uns noch einmal das Beispiel von oben ins Gedächtnis rufen, dann ist klar, dass nach dem Ende der Schleife die beiden Zeiger die Werte

$$\texttt{c} = \text{0xbffff80a} \qquad \text{und} \qquad \texttt{l} = \text{0xbffff800}$$

haben. Nach dem eben gesagten ergeben dann die Subtraktionen der Startadressen der Arrays

$$\begin{aligned} \texttt{c} - \texttt{char\_array} &= 10 \qquad \text{und} \\ \texttt{l} - \texttt{long\_array} &= 10, \end{aligned}$$

obwohl die tatsächliche Distanz in Byte zwischen den Zeigern im zweiten Fall viel größer ist als 10.

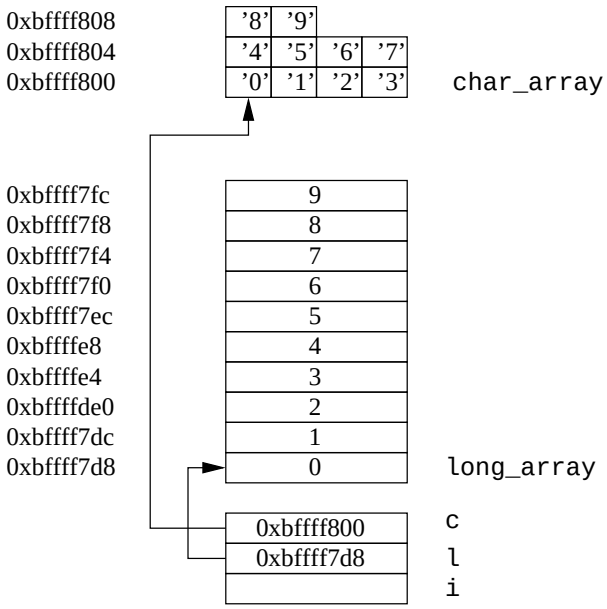


Abbildung A.2: Zeigerarithmetik: Speicher nach der Initialisierung der Variablen.

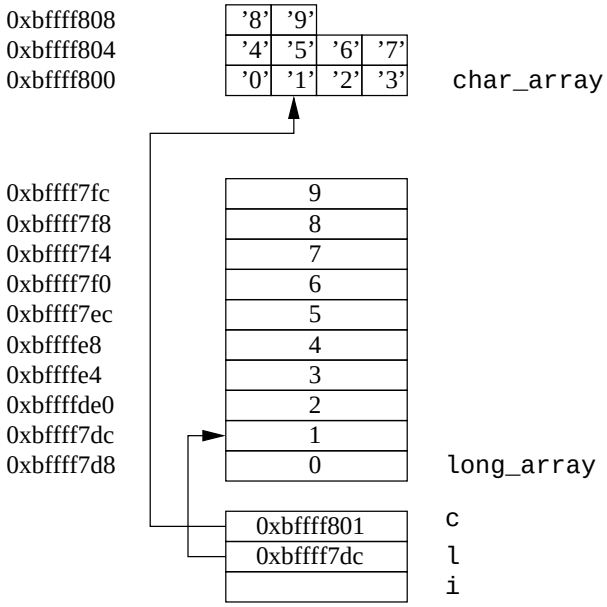


Abbildung A.3: Zeigerarithmetik: Speicher nach dem ersten Update von `c` und `l`.

Anhang B

Literaturhinweise

GoTo C Programmierung *G. Krüger*, Addison-Wesley
Das Buch ist ein ausführlicher C-Kurs sowohl für Einsteiger als auch Fortgeschrittene. Neben der Programmiersprache `C` werden auch noch Tools wie `make` oder Versionskontrollsysteme vorgestellt. Zu dem Buch gehört eine CD auf der sich `gcc` und `emacs` für MS-DOS bzw. Windows befinden.

C - kurz und gut *Kirch, Kirch-Prinz*, O'Reilly
Dieses Buch ist gut als Nachschlagewerk geeignet. Zum Erlernen von `C` selbst eignet es sich weniger. Hat man aber erstmal einige Grundfertigkeiten erlernt, dann kann man seine Kenntnisse durch „Stöbern" in diesem Buch gut vertiefen.

Programmieren in C *Kernighan, Ritchie*, Carl Hanser Verlag
Das Standardwerk über `C`. Zum Erlernen von `C` ist dieses Buch eher ungeeignet. Als Nachschlagewerk erfüllt es aber durchaus seinen Zweck.

http://www.csc.vill.edu/~lab/C/ Eine kurze Übersicht über die Programmiersprache `C` und einige der wichtigsten Bibliotheksfunktionen. Als Online-Referenz geeignet.

http://www.delorie.com Auf dieser Website gibt es für einige bekannte Linux/Unix-Programme wie z.B. `gcc` eine Windows©/MS-DOS©-Version.

Introduction to Algorithms *Cormen, Leiserson, Rivest, Stein*, The MIT Press
Dieses Buch ist das Standardwerk wenn es um eine Einführung in Algorithmen geht. Obwohl die Algorithmen, die in diesem Buch beschrieben werden, weit über das hinaus gehen, was in dieser Veranstaltung behandelt wird, so eignet es sich doch zum Nachschlagen oder selbständigen Weiterlesen.

Abbildungsverzeichnis

Tabellenverzeichnis