

# C-Programmierkurs

Jesko Hüttenhain  
rattle@uni-bonn.de

Lars A. Wallenborn  
lars@wallenborn.net

03730 AD

Once upon a time, man forged machine, from the heartless stone of the eastern desert. And Yawgmoth, great father of machines, spoke to man: "Ye doth be ruler of all machines, but not before ye hath mastered the Lega-C."

— *Lost Scriptures of Korja'Less*

## Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>4</b>
1.1	Der Speicher . . . . .	4
1.2	Maschinencode und Kompilierung . . . . .	5
1.3	Hello World . . . . .	6
1.4	Gcc unter Windows . . . . .	7
1.5	Architektur (32bit oder 64bit?) . . . . .	8
1.6	Java . . . . .	8
1.7	Cygwin . . . . .	10
1.7.1	PATH setzen . . . . .	10
1.8	Eclipse CDT . . . . .	12
1.9	Bedienung von Cygwin . . . . .	15
<b>2</b>	<b>Elementare Sprachkonstrukte</b>	<b>17</b>
2.1	Kommentare . . . . .	17
2.2	Variablen . . . . .	17
2.3	Numerische Konstanten . . . . .	19
2.4	Operatoren und Expressions . . . . .	20
2.5	Formalitäten: Statements und Expressions . . . . .	22
2.6	If-Else-Statement . . . . .	23
2.7	Logische- und Vergleichsoperatoren . . . . .	23
2.8	Der Schleifen erster Teil: while . . . . .	25
2.9	Der Schleifen zweiter Teil: for . . . . .	26

<b>3</b>	<b>Funktionen</b>	<b>27</b>
3.1	Funktionsdefinitionen . . . . .	27
3.2	Funktionsdeklaration vs. Funktionsdefinition . . . . .	29
3.3	Modulares Programmieren und Linken . . . . .	30
3.4	Der Präprozessor . . . . .	33
3.4.1	Makrodefinition . . . . .	33
3.4.2	Bedingte Texte . . . . .	34
3.4.3	Makrodefinition löschen . . . . .	35
3.5	Präprozessor - Compiler - Linker: Ein Beispiel . . . . .	35
<b>4</b>	<b>Adressierung und Arrays</b>	<b>39</b>
4.1	Adressen und Pointer . . . . .	39
4.2	Statische Arrays . . . . .	40
4.3	Pointerarithmetik . . . . .	41
4.4	Zeichenketten . . . . .	43
<b>5</b>	<b>Dynamische Speicherverwaltung</b>	<b>45</b>
5.1	Einleitung: Speicherverwaltung . . . . .	45
5.2	Allokierung von Speicher . . . . .	45
5.3	Speichermangel . . . . .	46
5.4	Speicher freigeben . . . . .	46
5.5	Speicherbereiche verändern . . . . .	47
5.6	Funktionen zum Speichermanagement . . . . .	48
5.7	Stringmanipulation . . . . .	49
<b>6</b>	<b>Eingabe und Ausgabe</b>	<b>51</b>
6.1	Einschub: Kommandozeilenargumente . . . . .	51
6.2	Dateien öffnen . . . . .	52
6.3	In Dateien schreiben . . . . .	53
6.4	Dateien lesen . . . . .	56
6.5	Byteweiser Dateizugriff . . . . .	59
6.6	Den Dateicursor verändern . . . . .	60
<b>7</b>	<b>Datenstrukturen</b>	<b>63</b>
7.1	Strukturdefinitionen . . . . .	63
7.2	Datenstrukturen . . . . .	65
7.3	Pointer auf Strukturen . . . . .	66
7.4	Intermezzo: Typdefinitionen . . . . .	67
7.5	Anwendung: Verkettete Listen . . . . .	69
<b>8</b>	<b>Weitere Sprachkonstrukte</b>	<b>72</b>
8.1	Bedingte Auswertung . . . . .	72
8.2	Konstante Variablen . . . . .	72

8.3	Funktionenpointer . . . . .	73
<b>9</b>	<b>Multithreading mit OpenMP</b>	<b>76</b>
9.1	Forking und Joining . . . . .	77
9.2	Sektionen . . . . .	78
9.3	Schleifen . . . . .	78
9.4	Barrieren . . . . .	79
9.5	Shared Memory . . . . .	80
9.6	single, master und critical . . . . .	82
9.7	Locks . . . . .	83
<b>A</b>	<b>Referenzen</b>	<b>85</b>
A.1	Referenz <math.h> . . . . .	85
A.2	Referenz <time.h> . . . . .	86
A.3	Referenz <stdlib.h> . . . . .	88
A.4	Referenz <limits.h> . . . . .	89
A.5	Referenz <float.h> . . . . .	89
<b>B</b>	<b>Operatorpräzedenzen</b>	<b>90</b>

# 1 Einführung

## 1.1 Der Speicher

Wenn wir von Speicher sprechen, so meinen wir nicht die Festplatte, sondern ein Bauteil des Computers, das während des laufenden Betriebs Daten nur für die Dauer eines Programmlaufs abspeichert. Man bezeichnet dies auch als RAM (Random Access Memory).

Der Speicher ist eine durchnummerierte Aneinanderreihung von Speicherzellen. Eine Speicherzelle ist ein elektronischer Chip, welcher wiederum 8 Bauteile enthält:

Diese Bauteile nennt man *Bits*. Ein Bit kann geladen und entladen werden, hat somit immer genau einen Zustand 1 oder 0. Jede Speicherzelle kann daher  $2^8 = 256$  Zustände annehmen (mögliche Kombinationen von Zuständen der einzelnen 8 Bits). Fast immer interpretiert man diese Zustände als ganze Zahlen zwischen 0 und 255. Diese Interpretation ist gegeben durch die Darstellung einer Zahl im Binärformat. Eine Speicherzelle bezeichnet man auch als *Byte*. Die Speicherzelle hat 8 ausgehende Drähte, auf welchen nur Strom fließt, wenn das dazugehörige Bit gesetzt (also 1) ist. Aus technischen Gründen kann immer nur ein ganzes Byte auf einmal gelesen oder neu beschrieben werden, keine einzelnen Bits.

Man möchte auch negative Zahlen in Bytes codieren können. Man könnte dafür das erste Bit als sogenanntes *Vorzeichenbit* reservieren, um sich zu merken, ob die Zahl positiv (Vorzeichenbit gleich 0) oder negativ (Vorzeichenbit gleich 1) ist. Die restlichen Bits können dann nur noch 128 verschiedene Zustände annehmen, also können wir nun die Zahlen von  $-127$  bis  $127$  darstellen. Dieses Prinzip zeigt anschaulich, dass es einen markanten Unterschied zwischen Daten und deren Interpretation gibt. Ein Byte kann als positive Zahl zwischen 0 und 255 oder aber als vorzeichenbehaftete Zahl zwischen  $-127$  und  $127$  interpretiert werden. Beides verwendet jedoch das gleiche Speichermedium. Man bezeichnet eine solche Interpretation als *Datentyp*. In der Realität wird zur Darstellung negativer Zahlen ein anderes Format, genannt „Zweierkomplement“, verwendet, welches praktischer zu implementieren ist und nur eine Null enthält (das obige Format hat eine  $+0$  und eine  $-0$ ).

Durch Zusammenschluss von Speicherzellen lassen sich auch größere Zahlen darstellen. Den Zusammenschluss von zwei Bytes bezeichnet man als *Word* (Wort), es kann

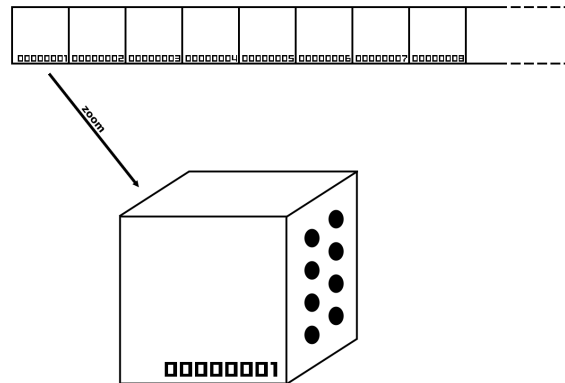


Abbildung 1: Der Speicher

bereits  $2^{16} = 65536$  Zustände annehmen. Ein *DWord* (Doppelwort) ist der Zusammenschluss von zwei Words und daher 4 Bytes oder 32 Bit lang. Es kann zum Speichern von Zahlen zwischen 0 und  $2^{32} - 1 = 4294967295$  verwendet werden. Dementsprechend bezeichnet man 64-Bit-Speicherblöcke als *QWord* (Quad Word). Eine Variable, die nur ein einzelnes Byte umfasst, wird gelegentlich auch als *char* bezeichnet, für „Character“. Der Name dieses Datentyps leitet sich daraus her, dass einzelne Buchstaben und andere Zeichen als Zahlen von 0 bis 255 im Computer abgespeichert werden. Zeichenketten und ganze Texte sind somit Speicherblöcke von  $n$  aufeinanderfolgenden Bytes (chars), wobei  $n$  die Länge der Zeichenkette ist.

Gelegentlich ist es nötig, auch über eine Darstellung reeller Zahlen zu verfügen. Dafür werden 8 Bytes Speicher (ein QWord) benötigt, die von einem internen Subprozessor als Kommazahlen interpretiert werden. Auf die genaue Realisierung werden wir nicht näher eingehen. Dieser Datentyp trägt den Bezeichner *double*.

## 1.2 Maschinencode und Kompilierung

Computer wurden ursprünglich als aufwendige Rechenmaschinen entworfen. Sie alle enthalten einen Kernchip, welcher auch heute noch alle tatsächlichen Berechnungen durchführt. Dieser Baustein ist die Central Processing Unit, auch kurz CPU. Die CPU enthält intern eine sehr geringe Anzahl Speicherzellen (etwa 8 bis 30), die auf modernen Computern für gewöhnlich die Größe eines QWords haben (obwohl auch noch DWords anzutreffen sind). Dies nennt man auch die *Registergröße* oder *Wortgröße* der CPU, die Speicherzellen selbst dementsprechend *Register*.

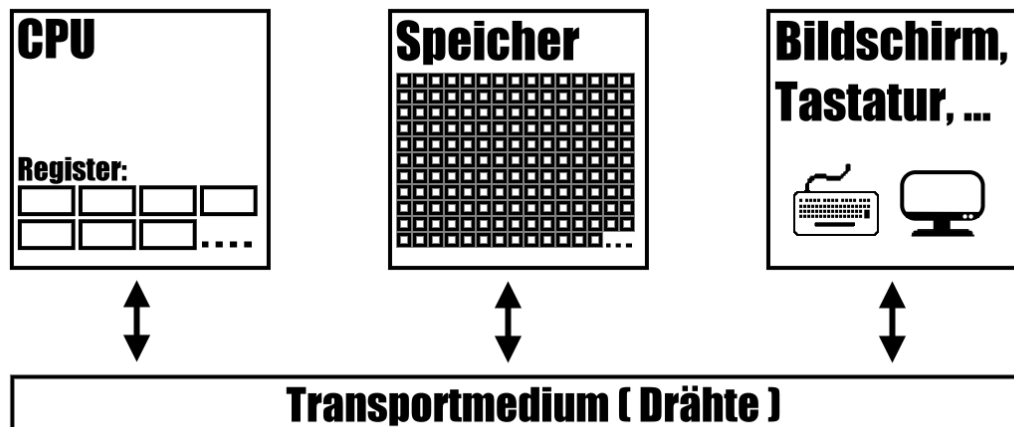


Abbildung 2: Schematischer Aufbau eines Computers

Die CPU eines Computers kann nur eine sehr geringe Anzahl von rudimentären Rechenoperationen durchführen. Genau wollen wir darauf nicht eingehen, doch besteht ein solcher CPU-Befehl beispielsweise daraus, den Inhalt zweier Register zu addieren, subtrahieren, multiplizieren, dividieren oder ähnliche arithmetische Operationen durchzuführen. Natürlich kann die CPU auch bis zu einer Registergröße Daten aus dem Speicher in ein Register laden, oder aus einem Register Daten in den Speicher schreiben. Jedem CPU-Befehl ist ein numerischer Code zugewiesen, welcher in einem Word gespeichert werden kann. Die so codierten CPU-Befehle heißen *Maschinencode*. Um ein Computerprogramm auszuführen, liest die CPU aus dem Speicher Maschinencode ein und führt die Befehle nacheinander aus. Es ist nun jedoch ausgesprochen mühsam, auf diese Art und Weise Algorithmen zu implementieren: Dies führte zur Entwicklung von Programmiersprachen, die eine für Menschen wesentlich zugänglichere Syntax vorweisen können. Als *Compiler* bezeichnet man Programme, die den Programmcode einer Programmiersprache in Maschinencode übersetzen. Diesen Vorgang nennt man Kompilierung. Der Compiler selbst muss freilich irgendwann mühsam als Maschinencode implementiert worden sein.

### 1.3 Hello World

Es ist Tradition, dass junge Schüler einer Programmierdisziplin sich Ihre Hörner an einem sogenannten Hello-World-Programm abstoßen. Ein solches Programm hat keinen Effekt, außer den Text „Hello World“ auf dem Computerbildschirm erscheinen zu lassen.

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

Listing 1: Ein Hallo-Welt-Programm in C

Wir können an dieser Stelle noch nicht genau auf die Bedeutung aller Programmierbefehle eingehen, wollen aber dennoch alles kommentieren. Die erste Zeile sorgt dafür, dass unserem Programm die Befehle zur Verfügung stehen, um Text auszugeben. Die nächste Zeile markiert den Einstiegspunkt des Programms, d.h. die Stelle, ab der beim Start später mit der Ausführung begonnen werden soll. Die auszuführenden Befehle sind in einem sogenannten *Block* zusammengefasst, welcher mit geschweiften Klammern umschlossen ist. Die Befehle selbst sind überschaubar: Der erste erzeugt die Ausgabe von „Hello World“ und der zweite beendet das Programm. Dabei wird der sogenannte *Fehlercode* 0 zurückgegeben, welcher signalisiert, dass beim Ausführen des Programms kein Fehler aufgetreten ist. Dieser Rückgabewert ist für den Anwender des Programms später

nicht erkennbar: er kann jedoch dazu dienen, verschiedene Programme miteinander kommunizieren zu lassen.

Außerdem bemerken wir an dieser Stelle, dass in C jeder *Befehl durch ein Semikolon beendet werden muss*. Dies ist eine wichtige Regel, deren Missachtung häufig zu scheinbar unerklärlichen Fehlern bei der Kompilierung führt. In der Tat dienen die Zeilenumbrüche im Quellcode „nur“ der Übersichtlichkeit, ein Befehl wird durch das abschließende Semikolon beendet. Daher wäre auch der folgende Quellcode zum obigen äquivalent und absolut korrekt:

```
1 #include <stdio.h>
2 int main() { printf("Hello World"); return 0; }
```


Listing 2: Hallo-Welt in einer Zeile

## 1.4 Gcc unter Windows

Der Compiler, mit dem wir unser Hello World - Programm und auch zukünftige Übungen in ausführbaren Maschinencode übersetzen werden, ist der C-Compiler aus der GNU Compiler Collection, welchen wir hier kurz exemplarisch einführen wollen. Er trägt den Namen *gcc*. Obgleich er ein sehr weit verbreiteter und gängiger Compiler ist, ist er selbstverständlich nicht der Weisheit letzter Schluss - es gibt eine Vielzahl weiterer Compiler, von denen einige leider nur käuflich zu erwerben sind.

Der gcc ist ein unter Linux entwickelter Compiler. Für eine ganze Sammlung von Linux-Programmen existieren Windows-Ports: Diese Sammlung heißt Cygwin. Wir werden hier kurz erläutern, wie Cygwin zu installieren und zu bedienen ist. Außerdem werden wir in der zweiten Woche des Kurses noch die Eclipse IDE mit den C/C++ Developer Tools und der Cygwin Toolchain verwenden. Die Installationsanleitung für Eclipse folgt hier ebenfalls, wenn ihr aber gerade den Kurs macht, müsst ihr erst einmal nur Cygwin installieren.

## 1.5 Architektur (32bit oder 64bit?)

Ihr solltet für die Installation wissen, ob ihr ein 32- oder 64bit Windows installiert habt. Wenn ihr das nicht bereits tut, könnt ihr es nachsehen, wenn ihr +R drückt und dort “control /name Microsoft.System” eingibt. Dort steht zum Beispiel “*System type: 64-bit Operating System*”.

Auf 64bit Betriebssystemen könnt ihr auch 32bit Software benutzen, aber nicht andersherum.

## 1.6 Java

Für Eclipse müsst ihr eine Java Runtime Environment installieren. Wenn ihr 32bit-Eclipse verwenden wollt, braucht ihr die 32bit Version der JRE. Wenn ihr 64bit-Eclipse verwenden wollt, braucht ihr die 64bit Version der JRE. Ihr könnt auch beide Varianten der JRE installieren. Ihr findet beide hinter folgendem Link: <http://www.java.com/en/download/>





## 1.7 Cygwin

Das Cygwin Setup könnt ihr auf <http://cygwin.org/> herunterladen.

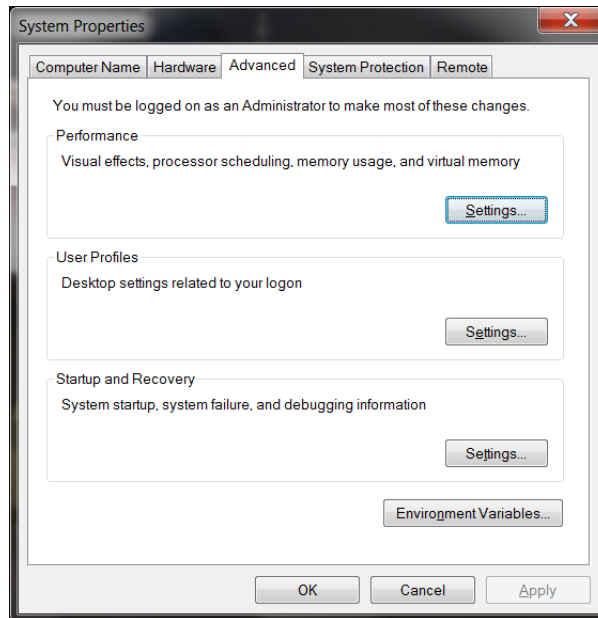


Es ist wichtig, Cygwin in einen Pfad ohne Leerzeichen o.ä. zu installieren. Belasst es also bitte bei dem empfohlenen Installationspfad `C:\cygwin` bzw. `C:\cygwin64`. Während der Installation wählt ihr bitte die folgenden Pakete aus:

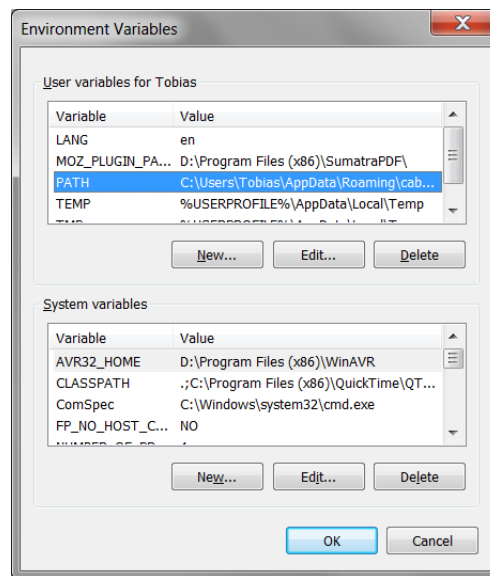
- gcc-core
- gdb
- make

### 1.7.1 PATH setzen

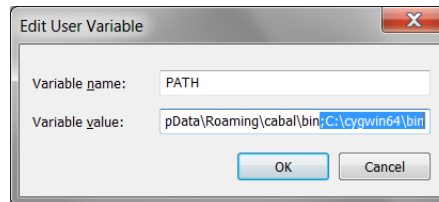
Danach müsst ihr Cygwin zu der Umgebungsvariable PATH hinzufügen, damit eclipse die benötigten Programme findet. Dafür drückt ihr **Win+R** und gebt "control sysdm.cp1„3" (sic) ein. Es öffnet sich ein Fenster *System Properties*.



Dort klickt ihr auf **Environment Variables**.

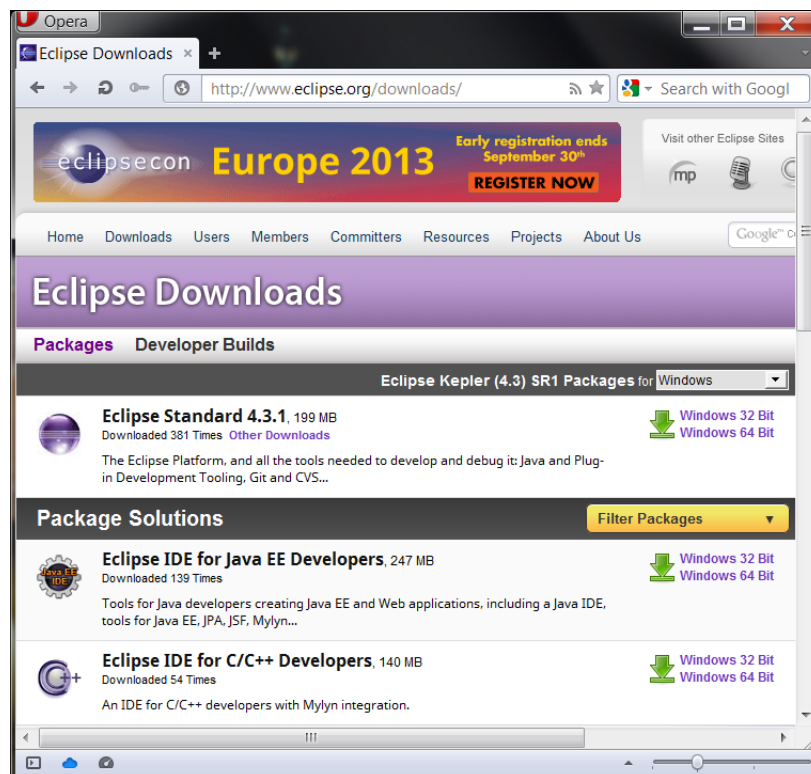


Schaut bitte oben unter *User variables* nach, ob dort bereits eine Variable mit dem Namen **PATH** existiert. Falls nicht, legt ihr eine neue Variable mit dem Namen **PATH** und dem Wert `"C:\cygwin64\bin"` an (falls ihr Cygwin in `C:\cygwin64` installiert habt, sonst passt den Pfad bitte entsprechend an). Falls die Variable bereits existiert, editiert ihr sie, bewegt den Cursor ganz ans Ende des Textes (z.B. indem ihr auf eure **Ende** Taste drückt) und fügt dort den gleichen Pfad ein, aber *mit einem Semikolon von dem existierenden getrennt*, also z.B. `";C:\cygwin64\bin"`.



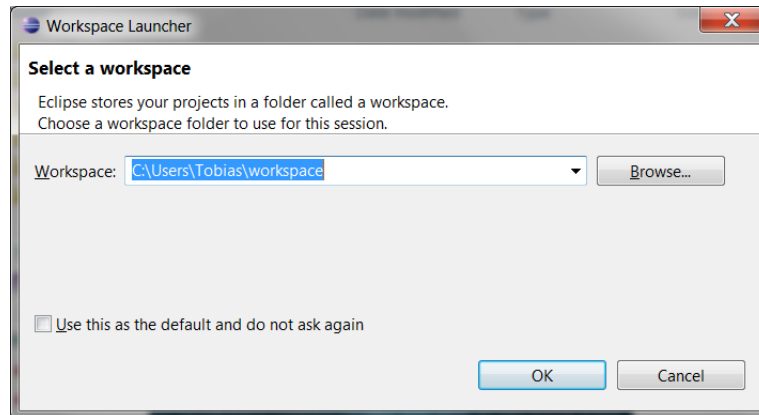
## 1.8 Eclipse CDT

Eclipse findet ihr auf <http://www.eclipse.org/downloads/>.

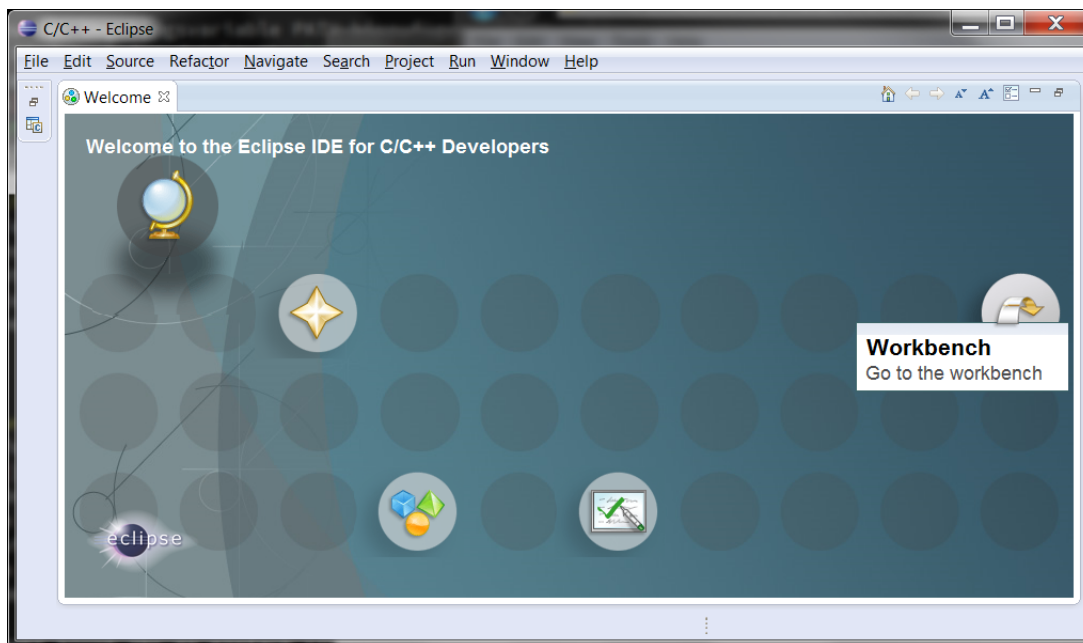


Ihr müsst dort die *Eclipse IDE for C/C++ Developers* herunterladen (und nicht etwa *Eclipse Standard*). Ladet die *.zip* Datei herunter und extrahiert sie an einen Ort eurer Wahl. Empfehlenswert ist "C:\Program Files" oder, wenn ihr 32bit-Eclipse auf einem 64-bit Betriebssystem verwendet, "C:\Program Files (x86)".

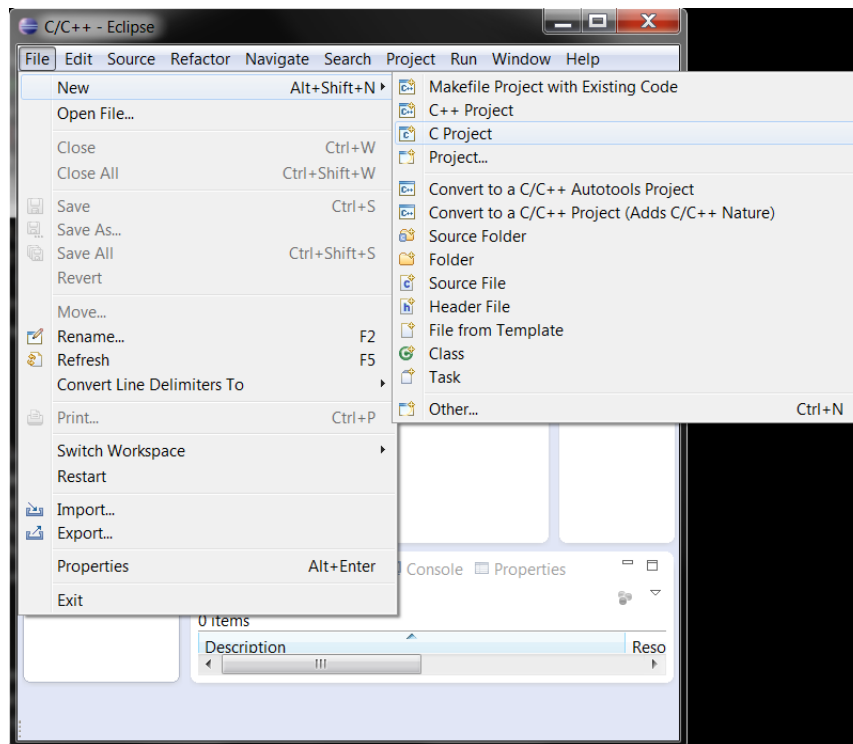
Nachdem ihr es entpackt habt könnt ihr es einfach starten. Legt euch einen Pfad für den Workspace *ohne Leerzeichen* an und klickt auf OK.



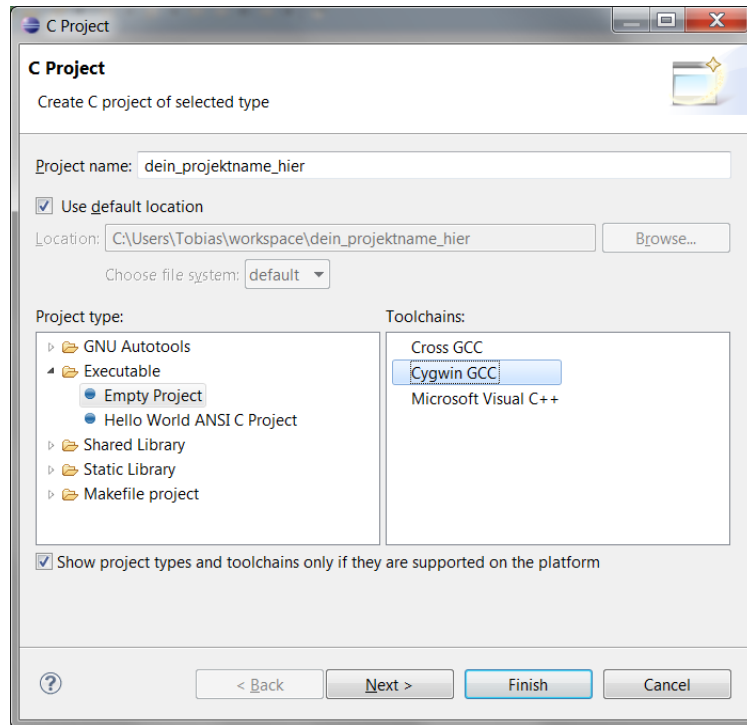
Auf dem *Welcome*-Bildschirm müsst ihr dann auf “Workbench” klicken (der Button rechts).





Legt dort ein neues C-Projekt an.



Wählt einen Projektnamen *ohne Leerzeichen*. Zum ersten Testen wählt das “Hello World ANSI C Project”, ansonsten “Empty Project”. Beim Hello-World Projekttyp wird direkt eine .c Datei generiert, die ihr ausprobieren könnt. Jetzt wählt die Cygwin GCC Toolchain aus (Vorsicht: wenn ihr danach den Projekttyp wechselt, ändert sich die gewählte Toolchain wieder).



Kompiliert das Projekt, indem ihr das Build-Icon  anklickt. Führt das kompilierte Programm aus, indem ihr auf  klickt.  
Viel Spaß!

## 1.9 Bedienung von Cygwin

Cygwin selbst lässt sich nun vom Startmenü aus aufrufen und präsentiert sich als schwarzes Fenster mit einer blinkenden Eingabe, etwa wie folgt:

```
rattle@lucy ~
$
```

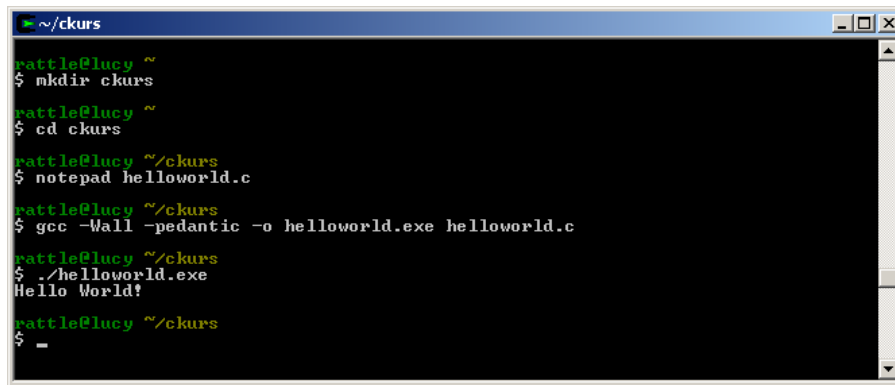
Hinter dem Dollarzeichen erwartet Cygwin nun einen Befehl. Es gibt zahlreiche Befehle, einige wichtige haben wir hier für euch aufgelistet:

Befehl	Effekt
ls	Listet den Inhalt des derzeitigen Verzeichnisses auf.
mkdir <name>	Erstellt einen Ordner mit dem angegebenen Namen
cd <ordner>	Wechselt in den angegebenen Ordner.
cp <quelle> <ziel>	Kopiert die Datei quelle nach ziel.
mv <quelle> <ziel>	Verschiebt die Datei quelle nach ziel.
rm <datei>	Löscht eine Datei.

Tabelle 1: Befehle der Cygwin-Kommandozeile

Ein einzelner Punkt steht für das derzeitige Verzeichnis und zwei Punkte für das darüberliegende. Der Befehl `cd .` hat also keinen Effekt und `cd ..` bewegt sich einen Ordner nach oben.

Darüber hinaus ist jedes Programm, das auf dem Computer (bzw. in Cygwin) installiert ist, ein Befehl. Durch eingabe von `notepad` beispielsweise öffnet sich der Windows-Texteditor und der Befehl `gcc` ruft den Compiler auf. Nun wollen wir unser Hello World Programm aus 1.3 kompilieren und ausführen:



```
~/ckurs
rattle@lucy ~
$ mkdir ckurs
rattle@lucy ~
$ cd ckurs
rattle@lucy ~/ckurs
$ notepad helloworld.c
rattle@lucy ~/ckurs
$ gcc -Wall -pedantic -o helloworld.exe helloworld.c
rattle@lucy ~/ckurs
$ ./helloworld.exe
Hello World!
rattle@lucy ~/ckurs
$ -
```

Abbildung 3: Kompilieren von “Hello World” unter Cygwin

Durch `notepad helloworld.c` erstellen wir die Textdatei `helloworld.c`. Es ist Konvention, dass Dateien, welche C-Quellcode enthalten, die Dateierdung `.c` erhalten. Wir bitten freundlich um Einhaltung dieser Konvention unter Androhung ritueller Enthauptung. Nach Tippen des oben angegebenen Quellcodes speichern wir die Datei und kehren zur Kommandozeile zurück. Der Befehl `gcc` hat folgendes Format:

```
gcc -Wall -Wpedantic -o EXECUTABLE QUELLDATEI
```

wobei in diesem Fall unsere Quelldatei den Namen `helloworld.c` trägt. Als Name für die Executable bietet sich der Name `helloworld.exe` an, doch natürlich steht einem die Entscheidung hier frei. Die Option `-Wall` ist eine Abkürzung für „Warning: All“ und bedeutet, dass der Compiler alle Warnungen ausgibt. Warnungen sind unser wichtigstes Hilfsmittel, um später Fehler in Programmen zu finden und zu beheben.

Nachdem wir den `gcc` aufgerufen haben, wurde im gleichen Verzeichnis eine Datei erstellt, die `helloworld.exe` heißt. Der Befehl `./helloworld.exe` besagt, dass die Datei `helloworld.exe` im derzeitigen Verzeichnis (der einzelne Punkt) ausgeführt werden soll.



## 2 Elementare Sprachkonstrukte

### 2.1 Kommentare

Obgleich Programmiersprachen gedacht sind, um dem Menschen verständlicher zu sein als der kryptische Maschinencode, können nur die wenigsten von uns C-Quellcode wie ein lustiges Taschenbuch lesen. Daher möchte man häufig an verschiedenen Stellen im Quellcode sogenannte *Kommentare* einfügen, d.h. Erläuterungen und Erklärungen zum Programm, welche nicht vom Compiler als Befehle interpretiert werden sollen. Um einen Kommentar zu beginnen, verwendet man die Zeichenfolge `/*` und beendet ihn durch die Zeichenfolge `*/`. Ein Beispiel:

```
1 /* HelloWorld v2.1a
2    (c) 2008 by Jesko & Lars */
3 #include <stdio.h>
4 int main() {
5     printf("Hello World\n");
6     return 0; /* Programm fehlerfrei beendet */
7 }
```

Listing 3: Hallo-Welt-Programm mit Kommentaren

### 2.2 Variablen

Ganz abstrakt ist ein Programm eine Maschinerie, die gewisse Daten erhält, und daraus neue Daten auf eine bestimmte Art und Weise berechnet. Daten treten in einem Programm stets in Form von sogenannten *Variablen* auf. Dabei ist eine Variable der Name für eine zusammenhängenden Region im Speicher des Computers, die durch ihren Datentyp eine Interpretation der dort gespeicherten Bits zugewiesen bekommt. Durch den Namen lässt sich im C-Programm die Speicherregion auslesen oder neu beschreiben. Der Programmierer kann sich zu Beginn eines Programmblocks wie folgt Variablen deklarieren (erstellen):

DATENTYP NAME = WERT;

Wann immer wir Definitionen wie oben angeben, so bedeutet ein unterstrichenes Wort, dass an dieser Stelle verschiedenes stehen kann. Für NAME etwa wird der *Name* eingefügt, welchen die Variable haben soll. Dies ist eine beliebige Zeichenfolge aus Buchstaben, Ziffern und Unterstrichen, welche nicht mit einer Ziffer beginnt. Der Name der Variablen sollte Aufschluss über ihren Zweck im Programm liefern. Variablennamen mit nur einem Buchstaben, obgleich in der Mathematik sehr verbreitet, sorgen bei Programmen in den meisten Fällen nur für Verwirrung. Ist ein Teil einer Definition grau gefärbt, so ist dieser Teil optional. Wir bemerken, dass das Semikolon oben nicht mehr optional ist.

Variablentyp	Deklaration	Ausgabebefehl
Ganzzahl	<code>int v;</code>	<code>printf("%i\n", v);</code>
Kleine Ganzzahl	<code>signed short int v;</code>	<code>printf("%hi\n", v);</code>
Große Ganzzahl	<code>signed long int v;</code>	<code>printf("%li\n", v);</code>
Ganzzahl $\geq 0$	<code>unsigned int v;</code>	<code>printf("%u\n", v);</code>
Kleine Ganzzahl $\geq 0$	<code>unsigned short int v;</code>	<code>printf("%hu\n", v);</code>
Große Ganzzahl $\geq 0$	<code>unsigned long int v;</code>	<code>printf("%lu\n", v);</code>
Byte (8 Bit)	<code>char c;</code>	<code>printf("%c\n", c);</code>
Kleine Fließkommazahl	<code>float f;</code>	<code>printf("%f\n", f);</code>
Große Fließkommazahl	<code>double d;</code>	<code>printf("%f\n", d);</code>

Tabelle 2: Die Funktion `printf` wird später genau erläutert werden

Tabelle 2.2 gibt Aufschluss über die zur Verfügung stehenden *Datentypen*, welche für `DATENTYP` eingesetzt werden können. Ein `int` beansprucht stets weniger oder genauso viel Speicher wie ein `long int` und stets mehr oder genauso viel Speicher wie ein `short int`. Die gewöhnliche Größe in Bits, die ein `int` belegt, hat sich im Laufe der Jahrzehnte von 16 über 32 zu mittlerweile 64 Bits gesteigert und könnte sich in der Zukunft weiter ändern.

Wir wollen noch etwas genauer verstehen, wie die verschiedenen ganzzahligen Datentypen zusammenhängen. Die Begriffe `signed` und `unsigned` sowie `short` und `long` sind bei der Deklaration einer `int`-Variablen optional. Wird einer der Ausdrücke nicht angegeben, so wird ein vom Computer und vom Betriebssystem abhängiger Standard gewählt. Sollte allerdings einer dieser Begriffe angegeben werden, so kann `int` selbst weggelassen werden, etwa so:

```
1 unsigned x;
```

Optional kann einer Variablen bereits bei der Deklaration ein *Wert* zugewiesen werden. Diesen Vorgang bezeichnet man als *Initialisierung* der Variablen. Beispiel:

```
1 unsigned long pi = 3; /* pi wird zu 3 initialisiert */
2 unsigned long x;      /* x ist undefiniert */
3 unsigned long y = pi; /* y wird zu pi initialisiert */
```

Achtung: Wird eine Variable nicht initialisiert, so ist sie *undefiniert*: Es ist unvorhersehbar, welchen Wert sie hat. Will man mehrere Variablen vom gleichen Typ deklarieren, so ist dies auch möglich, indem man sie nach Angabe des Datentyps lediglich durch Kommata trennt. Damit ist

```
1 unsigned long pi = 3, x, y = pi;
```

eine Kurzschreibweise für den Quellcode oben.

## 2.3 Numerische Konstanten

Ganzzahlige Konstanten sind uns bereits bei der Initialisierung von Variablen oben begegnet. Sie werden einfach als Folge von Ziffern in den Code eingegeben. Dabei können Zahlen zu unterschiedlichen Basen angegeben werden, nach folgender Regel:

1. Beginnt die Ziffernfolge *nicht* mit der Ziffer 0, so wird sie als “gewöhnliche” Zahlendarstellung im Dezimalsystem verstanden.
2. Andernfalls, wenn die Ziffernfolge mit 0x beginnt, so dürfen außer normalen Ziffern auch die Buchstaben A bis F in der Zahlendarstellung verwendet werden. Diese wird dann als *Hexadezimalzahl* (Darstellung zur Basis 16) interpretiert.
3. Andernfalls, wenn die Ziffernfolge mit 0 beginnt, wird sie als *Oktalzahl* (Darstellung zur Basis 8) verstanden. In diesem Fall sind die Ziffern 8 und 9 nicht erlaubt.

Dies ist insbesondere wichtig zu wissen, um Fehler zu vermeiden:

```
1 int x1 = 210; /*          x1 hat den Wert 210 */  
2 int x2 = 070; /* Achtung: x2 hat den Wert 56 */
```

Allerdings haben Konstanten auch einen Datentyp: Dieser Datentyp ist durch die Darstellung der Konstanten im Quellcode gegeben. Damit eine Konstante nicht als Ganzzahl, sondern als Fließkommazahl interpretiert wird, so muss sie einen Punkt zwischen den Vorkomma- und Nachkommastellen enthalten. Zusätzlich kann man wissenschaftliche Notation verwenden, was wir lediglich an einem Beispiel verdeutlichen wollen:

```
1 double pi = 3.141592653539793; /* Kommadarstellung für ~Pi */  
2 double c = 2.99792458e8;      /* wissenschaftliche Notation für  
    ~Lichtgeschwindigkeit */
```

Man kann Stellen vor und nach dem Punkt auch weglassen, diese werden dann automatisch zu 0. Beispiel:

```
1 double half = .5;
```

*Anmerkung:* Fließkommazahlen können ausschließlich als Darstellung zur Basis 10 angegeben werden. Führende Nullen werden bei der Angabe von Fließkommakonstanten einfach ignoriert.

## 2.4 Operatoren und Expressions

Eine *Expression* in C steht für einen Teil des Codes, welcher, ganz anschaulich ausgedrückt, einen Wert hat. Eine Variable ist beispielsweise bereits eine Expression, genau wie Konstanten.

Alle anderen Expressions in C entstehen aus Konstanten und Variablen durch deren Verknüpfung mittels *Operatoren* und Klammerung. Abstrakt ausgedrückt ordnet ein Operator einem oder mehreren Werten einen neuen Wert zu. So sind etwa alle Grundrechenarten

Operator	Expression	Wert der Expression
Addition	$a + b$	Summe von a und b
Subtraktion	$a - b$	Differenz von a und b
Multiplikation	$a * b$	Produkt von a und b
Division	$a / b$	Quotient von a und b
Modulo	$a \% b$	Rest einer Ganzzahldivision von a durch b

Tabelle 3: Arithmetische Operatoren

sogenannte *binäre* Operatoren (da sie zwei Werten einen Neuen zuweisen, nämlich gerade das Rechenergebnis). Beispiele für Expressions sind  $3+5*9$  und  $(pi+5)*9$ . Dabei gilt wie gewohnt: “Punkt- vor Strichrechnung”. Der Wert der Expression ist dann natürlich das Gesamtergebnis (beim ersten Beispiel also 48 und beim Zweiten 72). Wir werden im Laufe des Kurses außer den Grundrechenarten noch viele weitere Operatoren kennen lernen. Der Wert einer Expression kann durch den *Zuweisungsoperator* “=” in einer Variablen gespeichert werden:

```
1 pi = (pi+5)*9; /* setzt die Variable pi auf (pi+5)*9 */
```

Der Zuweisungsoperator entspricht also nicht dem mathematischen Gleichheitszeichen, sondern wird gelesen als “wird gesetzt auf”. Wer sich nun fragt, warum dies ein Operator sein soll, sei gesagt, dass eine Zuweisung in C auch einen Wert hat, nämlich gerade den Wert, der zugewiesen wird. Damit ist folgender Code korrekt:

```
1 x = pi = x+5*9; /* entspricht x = (pi=x+5*9); */
```

Hier wird also zunächst der Wert von  $(x+45)$  in der Variablen **pi** gespeichert – das Ergebnis dieser Zuweisungsoperation ist wiederum  $(x+45)$ , welches dann nach **x** geschrieben wird. Man sagt auch, der Zuweisungsoperator hat einen *Nebenefekt*, da er nicht nur einen Wert zurückgibt, sondern in Folge seiner Auswertung auch den Inhalt einer Speicherzelle verändert. Da jede Expression einen Wert hat, hat sie auch einen Datentyp. Gelegentlich möchte man durch Operatoren auch Expressions verknüpfen, die formal unterschiedliche Datentypen haben –

in diesem Fall muss eine der Expressions in eine Expression vom anderen Typ konvertiert werden. Diesen Vorgang nennt man *typecasting*. In vielen Fällen, wie etwa der Verknüpfung zweier Expressions mit Ganzzahltypen, nimmt C diese Konvertierung automatisch und meistens auch so vor, wie man es sich wünscht. Möchte man dennoch manuell eine Typkonvertierung durchführen, so geschieht dies durch folgende Syntax:

(DATENTYP) (EXPRESSION)

Als Beispiel könne man etwa eine Fließkommazahlen in eine Ganzzahl konvertieren, indem man schreibt:

```
1  double pi = 3.14159;
2  unsigned n = (unsigned) pi;
```

Die Konvertierung von Fließkommazahlen in Ganzzahlen geschieht durch Runden in Richtung 0. All dies wirft ein neues Licht auf die oben vorgestellten Rechenoperationen: Diese haben nämlich, abhängig vom Typ ihrer Argumente, eine unterschiedliche Arbeitsweise.

- Dividieren wir zwei Ganzzahlen, so wird eine Ganzzahldivision durchgeführt und der dabei entstehende Rest verworfen; also ergibt  $1/2$  den Wert 0 und  $7/3$  hätte den Wert 2. Durch explizites Typecasting lässt sich hier ein anderes Verhalten erzwingen schaffen:

```
1  unsigned x = 1, y = 2;
2  double half = (double)x/y; /* nun hat half den Wert 0.5 */
```

- Dividiert man eine Ganzzahl durch eine Fließkommazahl oder umgekehrt, so wird die Ganzzahl konvertiert und man erhält das (mehr oder minder) korrekte Ergebnis der Rechnung als Fließkommazahl.
- Generell gilt: Verknüpfen wir eine Fließkommazahl mit einer Ganzzahl, so wird diese in eine Fließkommazahl konvertiert, und das Ergebnis ist ebenfalls eine Fließkommazahl.

Es gibt nun noch einen weiteren nützlichen Rechenoperator, der bei einer Ganzzahldivision das Ergebnis verwirft und statt dessen den Rest als Ergebnis liefert: Der sogenannte *Modulo*-Operator, % (ein Prozentzeichen). So wäre etwa  $(6\%5)$  eine Expression mit dem Wert 1. Dieser Operator funktioniert nur mit Ganzzahlen.

Häufig hat man in der Programmierung Zuweisungen der Form  $a = a \times b$ , wobei  $\times$  einer der bisherigen, binären Rechenoperatoren ist. Dafür gibt es die Kurzschreibweise  $a \times= b$ . Ein Beispiel:  $a += 1$  würde den Wert von  $a$  um 1

erhöhen. Die Situation, eine Variable um zu de- oder inkrementieren, ergibt sich sehr häufig. Dafür verwendet man folgenden unären Operatoren .

Operator	Art	Wirkung	Wert der Expression
a++	postfix	inkrementiere a	a
++a	präfix	inkrementiere a	a+1
a--	postfix	dekrementiere a	a
--a	präfix	dekrementiere a	a-1

Tabelle 4: Kurzschreibweisen

*Anmerkung:* Es gibt Expressions, welche aufgrund ihrer Nebeneffekte nicht eindeutig sind, etwa `i=i+++i`. Diese Expression ist syntaktisch korrekt, doch es gibt keinen offiziellen Standard für ihren Wert. Man bezeichnet solche Expressions als *undefiniert*. Jeder Compiler hat bei derartigen Situationen das Recht, über die weitere Verfahrensweise zu entscheiden (Er könnte etwa die Expression auf eine mögliche Art und Weise auswerten oder einen Fehler erzeugen). Man sollte solche Expressions tunlichst vermeiden.

## 2.5 Formalitäten: Statements und Expressions

C-Programme setzen sich aus einer oder mehreren *Statements* zusammen. Wir haben bereits ein Statement kennen gelernt: Die Variablendeklaration. Außerdem kann man eine Expression zu einem Statement machen, indem man sie durch ein Semikolon abschließt. Ein Beispiel dafür ist die Zuweisung, die wir bereits in 2.4 kennen gelernt haben. Darüber hinaus kann man auch eine *Expressionliste* als Statement auswerten lassen: Dies ist eine durch Kommata separierte Liste von Expressions, welche durch ein abschließendes Semikolon zu einem Statement führt, in dem die Expressions der Reihe nach ausgewertet werden:

EXPRESSION 1, EXPRESSION 2, ..., EXPRESSION n;

Dies scheint zunächst nicht besonders nützlich zu sein, da wir die einzelnen Expressions durch Semikolons auch einzeln zu Statements machen können – im Zusammenspiel mit anderen Statements jedoch kann es sich als nützlich erweisen, mehrere Expressions als ein *einzelnes* Statement zusammenfassen zu können. In Wahrheit ist eine Expressionliste ebenfalls eine Expression: Das Komma ist ein binärer Operator , welcher seine beiden Argumente auswertet und den zweiten als Ergebnis liefert: Wert und Typ einer Expressionliste sind also immer Wert und Typ der letzten Expression in der Liste.

Ein weiteres, bereits bekanntes Statement ist der *Block*, welcher einfach mehrere Statements zu einem Statement zusammenfasst:

{ STATEMENT 1 STATEMENT 2 ... STATEMENT n }

Wichtig: Variablendeklarationen sind Statements, die nicht an jeder Stelle des Quellcodes verwendet werden dürfen. Variablendeklarationen müssen immer *die ersten Statements eines Blocks sein*. Die Variablen, die zu Beginn eines Blocks deklariert werden, gehören in gewisser Weise zu diesem Block. Nachdem der Block endet, werden die Variablen verworfen. Darüber hinaus kann eine Variablendeklaration in einem Block eine Variable des ihn umschließenden Blocks *überdecken*: Das heißt, in einem Block können Variablen deklariert werden, die außerhalb des Blocks bereits existieren. Diese beiden Variablen repräsentieren in diesem Fall *zwei unterschiedliche Speicherbereiche*, und innerhalb des Blocks können wir nur noch diejenige Variable verwenden, welche auch im Block deklariert wurde. Ein Beispiel:

```

1 #include <stdio.h>
2 int main() {                               /* Hier beginnt Block 1 */
3     int i = 4;
4     int j = 6;
5     {                                       /* Hier beginnt Block 2 */
6         int i=3;
7         printf("%i\n",i); /* Gibt 3 aus */
8         printf("%i\n",j); /* Gibt 6 aus */
9     }                                       /* Hier endet Block 2 */
10    printf("%i\n",i); /* Gibt 4 aus */
11 }                                          /* Hier endet Block 1 */

```

Listing 4: Beispiel für Überdeckung

## 2.6 If-Else-Statement

Einfache Rechenoperatoren erlauben uns nicht, komplexe Algorithmen zu implementieren – es fehlt die Möglichkeit, abhängig vom *Ergebnis* einer Operation *unterschiedlichen* Code auszuführen. Um dies zu ermöglichen, lernen wir nun das erste Programmierstatement kennen: Das If-Else-Konstrukt:

```

if ( BEDINGUNG ) BEFEHL 1
else BEFEHL 2

```

wobei die Bedingung eine beliebige Expression und die Befehle jeweils ein beliebiges Statement (meistens ein Block) sein können. Es wird der erste Befehl ausgeführt, sofern die Bedingung einen Wert ungleich 0 hat. Ansonsten, falls durch **else** angegeben, der zweite.

## 2.7 Logische- und Vergleichsoperatoren

Für die Bedingung im If-Else-Statement lernen wir noch einige weitere Operatoren kennen, die sogenannten *Vergleichsoperatoren*:

Operator	Syntax
Prüfen auf Gleichheit	<code>a == b</code>
Prüfen auf Ungleichheit	<code>a != b</code>
Prüfen, ob a echt größer als b ist	<code>a &gt; b</code>
Prüfen, ob a echt kleiner als b ist	<code>a &lt; b</code>
Prüfen, ob a größer oder gleich b ist	<code>a &gt;= b</code>
Prüfen, ob a kleiner oder gleich b ist	<code>a &lt;= b</code>

Tabelle 5: Vergleichoperatoren

Diese Operatoren liefern immer die Werte 1 oder 0, abhängig vom Ergebnis des Vergleiches. Damit wird das If-Else-Statement bereits zu einem mächtigen Werkzeug. Als Beispiel ein Codesegment, dass die Signumsfunktion für einen Eingabewert `x` implementiert:

```

1 if (x < 0)      /* falls x kleiner als 0 ist: */
2   y = -1;      /* das Signum ist -1 */
3 else          /* Ansonsten (falls x größergleich 0): */
4   y = (x != 0); /* falls x Null, wird y Null, sonst 1 */

```

Listing 5: Signumsfunktion

Um Vergleiche logisch zu verknüpfen, gibt es darüber hinaus auch noch die sogenannten *logischen Operatoren* (2.7). Dies sind ebenfalls binäre Operatoren (bis auf das logische Nicht), welche zwei Expressions die Werte 0 oder 1 zuweisen.

Operator	Syntax
Logisches Und	<code>A &amp;&amp; B</code>
Logisches Oder	<code>A    B</code>
Logische Verneinung	<code>!A</code>

Tabelle 6: Logische Operatoren

Die Ergebnisse der Logischen Operatoren lassen sich am einfachsten durch Wertetabellen veranschaulichen. Siehe dazu 2.7.

A	B	<code>A &amp;&amp; B</code>	<code>A    B</code>	<code>!A</code>
0	0	0	0	1
0	$\neq 0$	0	1	1
$\neq 0$	0	0	1	0
$\neq 0$	$\neq 0$	1	1	0

Tabelle 7: Logische Operatoren

Es gibt jedoch noch eine wichtige Eigenart dieser Operatoren zu erwähnen: Die logischen Operatoren werten nur so viele ihrer Argumente aus, bis das Ergeb-



nis der Verknüpfung bereits feststeht. So würde etwa bei der Auswertung von  $(1 \parallel x--)$  die Variable  $x$  *nicht* dekrementiert, da das Ergebnis der Operation bereits bei der Auswertung von  $1$  feststeht. Dies ist selbstverständlich nur von Bedeutung, sofern eine der auszuwertenden Expressions einen Nebeneffekt hat.

## 2.8 Der Schleifen erster Teil: while

Wollen wir einen bestimmten Codeblock mehrfach ausführen, so verwenden wir ein Statement, was als *Schleife* bezeichnet wird. Eine Schleife wiederholt die Befehle so lange, wie eine bestimmte Expression ungleich 0 ist. Die Syntax

```
while (BEDINGUNG) BEFEHL
```

weist den Computer an, zu prüfen, ob die Expression BEDINGUNG ungleich 0 ist. Ist dies der Fall, so wird das Statement BEFEHL ausgeführt und wir fangen wieder von vorne mit dem Prüfen der Bedingung an. Andernfalls wird die Schleife beendet. Meistens sollten die Befehle dafür sorgen, dass BEDINGUNG irgendwann zu 0 auswertet, indem etwa Variablen verändert werden. Man kann jedoch ebenso gut eine Endlosschleife programmieren:

```
1 while(1); /* leeres statement: tue nichts, und das für immer */
```

Listing 6: Endlosschleife

Wir wollen ein Beispiel angeben, welches die Geometrische Reihe  $\sum_{n=0}^{\infty} q^n = \frac{1}{1-q}$  ausrechnet:

```
1 double q = 0.2;
2 double x = 1.0, y = 0.0; /* Hilfsvariablen */
3 while (x > 1e-10) {      /* Solange x nicht zu klein ist */
4     y = y+x;              /* y speichert die Partialsummen */
5     x = x*q;              /* Berechne den nächsten Summanden */
6 }
7 /* Ergebnis steht jetzt in y */
```

Listing 7: Geometrische Reihe

Dieses Beispiel zeigt anschaulich, dass Programme deutlich aufwändiger sein können, als sie müssen. Wir hätten ebenso gut  $y=1./(1.-q)$ ; schreiben können, was der Computer in einem Bruchteil der Zeit berechnen könnte. Man sollte sich immer bemühen, nicht unnötig Rechenzeit zu vergeuden.

Wenn man das Statement BEFEHL gerne Ausführen möchte, bevor das erste Mal geprüft wird, ob BEDINGUNG zu 0 auswertet, so kann man eine do-while-Schleife verwenden:

```
do BEFEHL while(BEDINGUNG);
```

Man bemerke hier das zwingend erforderliche Semikolon am Ende.

## 2.9 Der Schleifen zweiter Teil: for

Die while-Schleife lässt sich verallgemeinern zur **for**-Schleife, dem folgenden Konstrukt:

```
for( INITIALISIERUNG; BEDINGUNG; STEP )  
    BEFEHL
```

wobei wir dies wie folgt durch eine while-Schleife modellieren könnten, sofern die Bedingung angegeben ist:

```
INITIALISIERUNG;  
while ( BEDINGUNG ) {  
    BEFEHL  
    STEP;  
}
```

Damit sind also die Initialisierung, der Step und die Bedingung jeweils eine Expression. Der Befehl ist, wie immer, ein einzelnes Statement (meistens ein Code-Block). Das Beispiel aus dem letzten Abschnitt kann man also so umschreiben:

```
1 double x,y,q = 0.2;  
2 for (x=1.,y=0.; x>1e-10; x = x*q)  
3   y = y+x;
```

Listing 8: Geometrische Reihe mit einer For-Schleife

Lässt man bei der **for**-Schleife die Bedingung weg, bricht die Schleife nicht ab. Genauer: Die Schleife verhält sich so, als wäre die Bedingung die konstante Expression 1. Step oder Initialisierung sind ebenfalls optional und können weggelassen werden – also ist folgende Schleife eine Endlosschleife: **for(;;)**;

Es gibt zwei besondere Statements, welche innerhalb von Schleifen verwendet werden können:

Statement	Effekt
<b>break;</b>	Schleife abbrechen bzw. zum nächsten Statement nach der Schleife springen. In einer <b>for</b> -Schleife wird der Step noch einmal ausgeführt.
<b>continue;</b>	Nur diesen Schleifendurchlauf abbrechen (zum Step springen).

Tabelle 8: Spezielle Schleifenbefehle

Bei einer **for**-Schleife sorgt ein **continue**-Statement also dafür, dass der Step noch ausgeführt wird, bevor die Bedingung abgefragt wird und dann evtl. der nächste Schleifendurchlauf beginnt.

## 3 Funktionen

Funktionen sind ein grundlegendes und wichtiges Konzept der Programmierung. Sie ermöglichen es, häufig benötigte Programmzeilen als “Unterprogramm” zusammenzufassen. An anderen Stellen im gleichen Programm kann man dann durch einen sogenannten *Aufruf* der Funktion dorthin verzweigen. In der Funktion selbst kann man durch das `return` - Statement dafür sorgen, dass die Ausführung an der Stelle fortgesetzt wird, an der die Funktion aufgerufen wurde. Wie ihre mathematischen Äquivalente können Funktionen Argumente erhalten und einen Rückgabewert besitzen.

### 3.1 Funktionsdefinitionen

Eine *Funktionsdefinition* hat folgende Form:

```
RÜCKGABETYP FUNKTIONSNAME(  
    PARAMETERTYP 1 PARAMETERNAME 1,  
    PARAMETERTYP 2 PARAMETERNAME 2,  
    ...,  
    PARAMETERTYP n PARAMETERNAME n ) {  
    BEFEHLE  
}
```

Der Rückgabetyt ist hierbei ein beliebiger Datentyp - dieser bestimmt, welchen Datentyp der Ausdruck des Funktionsaufrufes hat. Ein *Funktionsaufruf* hat die Syntax:

```
FUNKTIONSNAME ( PARAMETER 1, ..., PARAMETER n )
```

Dies bedeutet, dass eine Funktion ein vom Programmierer neu definierter Operator ist: Sie weist einem oder mehreren Werten einen neuen Wert (den Rückgabewert) zu.

Bei jedem Funktionsaufruf werden zunächst neue Variablen PARAMETERNAME 1 bis PARAMETERNAME n erstellt, welche vom in der Funktionsdefinition angegebenen Datentyp sind. Dann werden die Expressions PARAMETER 1 bis PARAMETER n ausgewertet und den Variablen in der entsprechenden Reihenfolge zugewiesen. Anschließend werden die Befehle in der Funktionsdefinition ausgeführt, bis der Wert berechnet wurde, den der Funktionsaufruf haben soll. Durch das folgende Statement beendet die Funktion sich selbst augenblicklich und legt ihren sogenannten *Rückgabewert* fest: Der Wert des Funktionsaufrufes.

```
return RÜCKGABEWERT;
```

Die Parameter in der Funktionsdefinition sind Variablendeklarationen, deren Initialisierung durch den Funktionsaufruf statt findet. Sie gehören zum Block der Funktionsdefinition und können (sollten) dort zur Berechnung des Rückgabewerts verwendet werden – Dennoch kann eine Funktion selbstverständlich zu Beginn weitere, interne Variablen erstellen.

Innerhalb der Funktion sind dies aber insgesamt die *einzigsten* Variablen, auf die direkt (mit Namen) zugegriffen werden kann. Wir wollen nun Code für eine Funktionsdefinition vorstellen, welche das Signum einer Ganzzahl ausrechnet (siehe auch 2.7) und diese Funktion dann aufrufen:

```
1 #include <stdio.h>
2
3 int sign( int x ) {
4     if (x < 0) return -1;
5     else return (x != 0);
6 }
7
8 int main() {
9     printf("%i\n", sign(-5)); /* Wird -1 ausgegeben. */
10    return 0;
11 }
```

Listing 9: Funktionsdefinition und -aufruf

Noch ein Beispiel:

```
1 #include <stdio.h>
2
3 /* berechnet zahl hoch exponent */
4 double potenz(double zahl, unsigned int exponent) {
5     double ergebnis;
6     for (ergebnis = 1.0; exponent; exponent--)
7         ergebnis *= zahl;
8     return ergebnis;
9 }
10
11 int main() {
12     printf("%f\n", potenz(0.5, 4) ); /* Wird 0.0625 ausgegeben. */
13     return 0;
14 }
```

Listing 10: Funktion zum Berchnen ganzer Potenzen von Fließkommazahlen

Wir können nun zum ersten mal feststellen, welche genaue Form ausführbarer C-Quellcode hat: Dieser setzt sich nämlich aus Funktionsdefinitionen zusammen, welche wiederum aus Statements bestehen, die bei Aufruf der Funktion in angegebener Reihenfolge ausgeführt werden. Es muss eine Funktion mit dem Namen `main` geben, welche zu Beginn des Programms gestartet wird.

Wir lernen an dieser Stelle noch einen neuen Datentyp kennen, den Datentyp `void`. Man kann keine `void`-Variablen deklarieren, denn eine Expression mit Datentyp `void` hat *keinen* Wert. Allerdings gibt es Funktionen mit *Rückgabety*p `void`, welche man auch als Prozeduren bezeichnet. Eine Prozedur muss kein `return`-Statement enthalten, kann jedoch das leere `return`-Statement `return;` verwenden, um sich selbst zu beenden.

```
1 #include <stdio.h>
2
3 void printInt(int x) {
4     printf("%i\n", x);
5 }
6
7 int main() {
8     printInt(42); /* gebe 42 auf der Kommandozeile aus */
9     return 0;
10 }
```

Listing 11: Beispiel für eine Prozedur

## 3.2 Funktionsdeklaration vs. Funktionsdefinition

Möchte man eine Funktion aufrufen, so muss die Definition dieser Funktion im Quellcode vor dem Funktionsaufruf liegen, da der Compiler die aufzurufende Funktion bereits “kennen” muss, damit er einen Aufruf korrekt in Maschinencode übersetzen kann: Dazu muss er wenigstens wissen, wie genau die Funktionsargumente und der Rückgabety

Man kann diese Informationen jedoch angeben, bevor man die Funktion tatsächlich definiert, indem man lediglich eine *Funktionsdeklaration* verwendet. Dieses Statement sieht wie folgt aus:

```
RÜCKGABETYP FUNKTIONSNAME(
    PARAMETERTYP 1 PARAMETERNAME 1,
    ...,
    PARAMETERTYP n PARAMETERNAME n );
```

Die Deklaration enthält also nur den sogenannten *Funktionskopf*, in dem alle für den Compiler wichtigen Informationen enthalten sind. Nachdem die Funktion deklariert ist, kann man sie im nachfolgenden Quellcode verwenden. An irgendeiner Stelle muss allerdings dann die tatsächliche Definition stehen. Hier ein Beispiel, welches ohne dieses Sprachkonstrukt gar nicht möglich wäre:

```

1 #include <stdio.h>
2
3 /* Funktionsdeklarationen */
4 int ungerade(int); /* diese Deklaration ist notwendig. */
5 int gerade(int); /* diese Deklaration nicht, ist aber huebsch. */
6
7 /* Funktionsdefinitionen */
8 int gerade(int n) {
9     /* testet, ob n gerade ist */
10    if (n == 0) return 1;
11    else return ungerade(n-1); /* wir müssen "ungerade" kennen */
12 }
13 int ungerade(int n) { /* testet, ob n ungerade ist */
14    if (n == 0) return 0;
15    else return gerade(n-1);
16 }
17
18 int main() {
19     if ( gerade(5) ) {
20         printf("Verkehrte Welt\n");
21         return 1;
22     } else return 0;
23 }

```

Listing 12: Funktionsdeklarationen sind notwendig

Die Umsetzung dieser Funktionen ist natürlich haarsträubend ineffizient, umständlich und unverständlich. Wir konnten jedoch kein Besseres Beispiel für Funktionen finden, die sich auf diese Art und Weise gegenseitig aufrufen: Man bezeichnet dies auch als *indirekte Rekursion*.

### 3.3 Modulares Programmieren und Linken

Die Kompilierung von großen Programmen zu schnellem und effizientem Maschinencode bedarf eines deutlich merkbaren Rechenaufwands. Während der Weiterentwicklung oder Fehleranalyse solcher Programme müssen allerdings ständig Teile des Programmcodes verändert werden und es wäre zu zeitaufwändig, das gesamte Programm ständig neu zu kompilieren - insbesondere, da sich ja nur gewisse Teilbereiche des Programms ändern - etwa nur eine bestimmte Funktion. Man geht deswegen dazu über, einzelne Teile eines Programms so voneinander zu trennen, dass der Compiler sie unabhängig voneinander in Maschinencode übersetzen kann. Diese Teile nennt man auch *Module*.

Nachdem ein solches Modul kompiliert wurde, ist es natürlich kein lauffähiges Programm - insbesondere verwendet das Modul unter Umständen Funktionen, deren Programmcode sich in anderen Modulen befindet. Um diese Abhängigkeiten aufzulösen, wird in der Schlussphase der Codegenerierung ein Programm (der *Linker*) gestartet, um die kompilierten Module zu einem lauffähigen Programm

zusammenzufügen. Diesen Vorgang bezeichnet man dementsprechend als *Linken*. Ein Modul in C ist zunächst eine Datei mit Dateiendung “c”. Jede solche .c-Datei wird von dem Compiler zu einer sogenannten *Objektdatei* kompiliert, welche das kompilierte Modul darstellt. Diese Objektdatei enthält Informationen darüber, welche Funktionen das Modul enthält und welche Funktionen von dem Modul aus anderen Modulen benötigt werden. Sind einmal alle Objektdateien erstellt, löst der Linker die Abhängigkeiten zwischen ihnen auf und fügt die Objektdateien zu einem lauffähigen Programmcode zusammen. Dieser Vorgang ist unabhängig von der Kompilierung.

Bei der Kompilierung ist es jedoch erforderlich, dass Funktionen definiert werden, bevor sie im Quellcode danach verwendet werden. Existiert etwa eine Quellcode-datei `moremath.c`, welche unter anderem eine Funktion

```
1 unsigned fibonacci(unsigned n)
```

beinhaltet, so könnte man die folgende `main.c` natürlich trotzdem nicht erfolgreich kompilieren, da zumindest eine Deklaration der Funktion fehlt:

```
1 #include <stdio.h>
2 /* Hier fehlt eine Deklaration oder Ähnliches */
3 int main() {
4     unsigned j;
5     for (j=1; j<10; j++)
6         printf("%u\n", fibonacci(j));
7     return 0;
8 }
```

Listing 13: Fehlende Deklaration

Man mache sich klar, dass dies ein Problem des Compilers und völlig unabhängig vom Linker ist. Um dieses Problem zu lösen, gehört zu jedem Modul auch eine *Headerdatei* mit der Dateiendung “h”, welche den gleichen Namen wie die Quellcode-datei des Moduls erhält. Diese enthält nur Funktionsdeklarationen. Im Sinne des obigen Beispiels sähe die Headerdatei `moremath.h` etwa so aus:

```
1 unsigned faculty(unsigned n); /* berechnet n! */
2 unsigned fibonacci(unsigned n); /* berechnet die n-te Fibonaccizahl
   */
```

Listing 14: Header-Datei für das `moremath`-Modul

Also enthält die Headerdatei lediglich Informationen über die Verwendung der Funktionen, die sich im zugehörigen Modul befinden, damit eine Kompilierung mit voneinander getrenntem Code überhaupt erst möglich wird. Mit dieser Datei ist `main.c` in folgender Variante nun kompilierbar:

```

1 #include <stdio.h>
2 #include "moremath.h"
3 int main() {
4     unsigned j;
5     for (j=1; j<10; j++)
6         printf("%u\n", fibonacci(j));
7     return 0;
8 }

```

Listing 15: Deklaration fehlt nun nicht mehr

Die Headerdateien von selbstgeschriebenen Modulen werden durch die `#include` - Anweisung direkt in den Quellcode eingefügt (kopiert) . Die Headerdateien eigener Module werden mit Anführungszeichen angegeben, Headerdateien von Systemmodulen mit spitzen Klammern. In der Tat gibt es bereits im System vorhandene Modul wie etwa `stdio` und `math`, welche sich in ihrer Funktionsweise nicht von selbst erstellten Modulen unterscheiden. Das Modul `moremath.c` könnte nun wie folgt aussehen:

```

1 #include "moremath.h"
2
3 unsigned faculty(unsigned n) {
4     unsigned f = 1;
5     for (;n;n--) f *= n;
6     return f;
7 }
8
9 unsigned fibonacci(unsigned n) {
10     if (n < 2) return 1;
11     else return fibonacci(n-1) + fibonacci(n-2);
12 }

```

Listing 16: Das moremath-Modul

Die Quellcodedatei bindet für Gewöhnlich ihre zugehörige Headerdatei ein. Dies hat viele Vorteile, die in Zukunft noch klarer werden, doch einen Grund kennen wir bereits: Sollten die Funktionen eines Moduls sich gegenseitig verwenden, so vermeiden wir durch Einfügen aller Deklarationen zu Anfang Compilerfehler.

Zusammenfassung: Der Compiler ist während der Kompilierung lediglich auf vollständige Deklarationen aller verwendeten Funktionen angewiesen. Diese befinden sich in den jeweiligen Headerdateien. Ist die Kompilierung abgeschlossen, muss der Linker aus einer Menge von kompilierten Modulen ein Programm erstellen. Dazu sucht er zunächst das Modul, welches die `main` Funktion enthält, da an dieser Stelle die Ausführung des Programms beginnen soll. Von diesem Modul ausgehend sucht der Linker nun zu jedem noch nicht verknüpften Funktionsnamen in allen Modulen (auch den Systemmodulen) nach einer Funktion mit dem gleichen Namen und bindet jenes Modul ein, sobald er es gefunden hat. Dies wird



fortgeführt, bis alle Namen aufgelöst sind und ein lauffähiges Programm erstellt werden kann.

Es sei an dieser Stelle noch einmal betont, dass das Konzept von Headerdateien (`.h`) ein Modul auf Compilerebene beschreibt, während die Aufteilung von Funktionen auf verschiedene Quellcodedateien (`.c`) ein Modul auf Linkerebene beschreibt. Diese beiden Konzepte funktionieren unabhängig voneinander. Eine Headerdatei könnte etwa Deklarationen von Funktionen enthalten, die auf zwei Quellcodedateien verteilt sind, oder man könnte Deklarationen von Funktionen einer Quellcodedatei auf mehrere Headerdateien verteilen. Auch die Namen von Header- und Quellcodedatei eines Moduls *müssen* streng genommen nicht übereinstimmen - all dies gebietet nur der gute Stil und die Übersichtlichkeit des gesamten Projekts.

### 3.4 Der Präprozessor

Bevor der Compiler tatsächlich mit der Kompilierung eines C-Programms beginnt, wird ein Programm aufgerufen, dass als Präprozessor bezeichnet wird. Er führt ausschließlich Textersetzungen im Quellcode durch. Er kann durch spezielle Befehle im Quellcode gesteuert werden, welche durch eine führende Raute (`#`) gekennzeichnet werden. Einige dieser Befehle kennen wir bereits, etwa geschieht das Einbinden von Headerdateien durch den Präprozessorbefehl:

```
1 #include <stdlib.h>
2 #include "myheader.h"
```

Listing 17: Einbinden von Header-Dateien sind Präprozessoranweisungen

Hier erfolgt eine reine Textersetzung - der Inhalt der Datei `myheader.h` wird vollständig an die Stelle des `include` - Befehls kopiert. Die spitzen Klammern sind notwendig, um eine Standardheader einzufügen, während Anführungszeichen verwendet werden, um selbst erstellte Header-Dateien einzufügen. Es gibt jedoch noch einige weitere nützliche Präprozessorbefehle.

#### 3.4.1 Makrodefinition

```
#define MAKRO REPLACE
```

ist eine sogenannte Makrodefinition. Sie weist den Präprozessor an, die Zeichenkette `MAKRO` im Folgenden immer durch `REPLACE` zu ersetzen. Dabei kann `REPLACE` auch der leere String sein bzw. weggelassen werden. Dies kann etwa dazu genutzt werden, Konstanten zu definieren:

```
1 #define PI 3.1415926535897931
```

Es gibt weiterhin die Möglichkeit, einem Makro Parameter zu übergeben, die in `REPLACE` verwendet werden können:

```
1 #define SQUARE(_x)  ((_x)*(_x))
```

Ein Auftreten von `SQUARE(3)` im Quellcode würde an dieser Stelle den String `((3)*(3))` einfügen. Diese Makros sollten mit Vorsicht genossen werden, da lediglich Textersetzungen durchgeführt werden. Ist etwa `funct` eine langsame Funktion, so führt die Verwendung von `SQUARE(funct(x))` zu `((funct(A))*(funct(A)))`. Dies bedeutet, dass die Funktion unnötigerweise zwei mal aufgerufen wird. Ähnlich führt `SQUARE(x--)` dazu, dass die Variable `x` zwei mal dekrementiert wird. Man mag sich weiterhin wundern, warum bei der Definition von `SQUARE` so viele Klammern verwendet wurden, doch man führe sich einfach vor Augen, dass `SQUARE(2+2)` ohne die inneren Klammern durch `(2+2*2+2)` ersetzt würde. Es ist sinnvoll, die Parameter bei Makrodefinitionen mit einem Unterstrich zu beginnen, damit keine Konflikte mit tatsächlich vorhandenen Variablen entstehen können.

### 3.4.2 Bedingte Texte

```
#if AUSDRUCK
    TEXT A
#else
    TEXT B
#endif
```

Dieser Befehl erlaubt es uns, mit dem Präprozessor kleinere Fallunterscheidungen durchzuführen. Wenn die Bedingung der `if` - Anweisung erfüllt ist, so wird Text A eingefügt, andernfalls Text B. Der `else` - Zweig der Anweisung ist optional. Auf die verschiedenen Möglichkeiten für Ausdrücke lohnt es sich kaum, hier einzugehen - der wichtigste Ausdruck ist vermutlich

```
#if defined(MAKRONAME)
```

welcher prüft, ob ein Makro mit Namen `MAKRONAME` bereits definiert ist. Damit lassen sich insbesondere Inklusionskreise bei Headerdateien vermeiden:

```
1 #if !defined(MYMATH_H)
2 #define MYMATH_H
3 /* Inhalt */
4 #endif
```

Listing 18: Zirkuläre Inclusion verhindern

Beim ersten Einfügen dieser Datei mittels `#include` wird das Makro `MYMATH_H` noch unbekannt sein, daher wird der Präprozessor den Text nach `#if` einfügen und insbesondere das Makro `MYMATH_H` definieren. Sollte die Datei ein zweites mal per `#include` eingefügt werden, ist das Makro `MYMATH_H` nun definiert und

der Präprozessor überspringt alles zwischen `#if` und `#endif`. Damit ist also sichergestellt, dass der Inhalt einer Headerdatei nur ein einziges Mal in einem Projekt eingefügt wird. Man nennt dieses Konstrukt auch *Include Guards* (Include-Wächter). Es sollte nach Möglichkeit bei allen Headerdateien verwendet werden, da der Präprozessor sonst in eine Endlosschleife gerät, sobald zwei Headerdateien sich gegenseitig per `#include` einbinden.

Da dieser Befehl überaus nützlich und weit verbreitet ist, gibt es eine Kurzschreibweise:

```
#ifndef MYMATH_H ⇒ #if !defined(MYMATH_H)
#define MYMATH_H ⇒ #if defined(MYMATH_H)
```

### 3.4.3 Makrodefinition löschen

`#undef MAKRONAME`

Wird verwendet, um ein bereits definiertes Makro zu löschen. Ist das angegebene Makro noch nicht definiert, hat der Befehl keine Auswirkung.

## 3.5 Präprozessor - Compiler - Linker: Ein Beispiel

Wir wollen anhand eines bereits bekannten Beispiels (mit Bildern) den Werdegang eines Projekts aus Quellcodedateien zur fertigen, ausführbaren Datei illustrieren. Angenommen also, wir hätten das folgende Projekt:

```
1 #include <stdio.h>
2 #include "gmodul.h"
3
4 int main() {
5     if (gerade(5)) {
6         printf("Verkehrte Welt\n");
7         return 1;
8     } else return 0;
9 }
```

Listing 19: main.c

```
1 #ifndef _UMODUL_H
2 #define _UMODUL_H
3 #include "gmodul.h"
4 int ungerade(unsigned n);
5 #endif
```

Listing 20: umodul.h

```

1 #include "umodul.h"
2 int ungerade(unsigned n) {
3     if (n==0) return 0;
4     else return gerade(n-1);
5 }

```

Listing 21: umodul.c

```

1 #ifndef _GMODUL_H
2 #define _GMODUL_H
3 #include "umodul.h"
4 int gerade(unsigned n);
5 #endif

```

Listing 22: gmodul.h

```

1 #include "gmodul.h"
2 int gerade(unsigned n) {
3     if (n==0) return 1;
4     else return ungerade(n-1);
5 }

```

Listing 23: gmodul.c

Durch Aufruf von

```
gcc -Wall -o project5.exe main.c gmodul.c umodul.c
```

wollen wir das Projekt in eine ausführbare Datei übersetzen. Was geschieht in den einzelnen Phasen? Wären die Präprozessorbefehle in den ersten beiden und der letzten Zeile von `gmodul.h` bzw. `umodul.h` nicht vorhanden, so würde der Präprozessor beim Einfügen von `umodul.h` zunächst `gmodul.h` einfügen und dabei auf Anweisung treffen, `umodul.h` einzufügen - eine Endlosschleife. So aber erzeugt der Präprozessor folgende, überarbeitete Quellcodedateien:

```

1 /* inhalt von stdio.h */
2 int ungerade(unsigned n);
3 int gerade(unsigned n);
4
5 int main() {
6     if (gerade(5)) {
7         printf("Verkehrte Welt\n");
8         return 1;
9     } else return 0;
10 }

```

Listing 24: main~.c

```

1 int gerade(unsigned n);
2 int ungerade(unsigned n);
3
4 int ungerade(unsigned n) {
5     if (n==0) return 0;
6     else return gerade(n-1);
7 }

```

Listing 25: umodul~.c

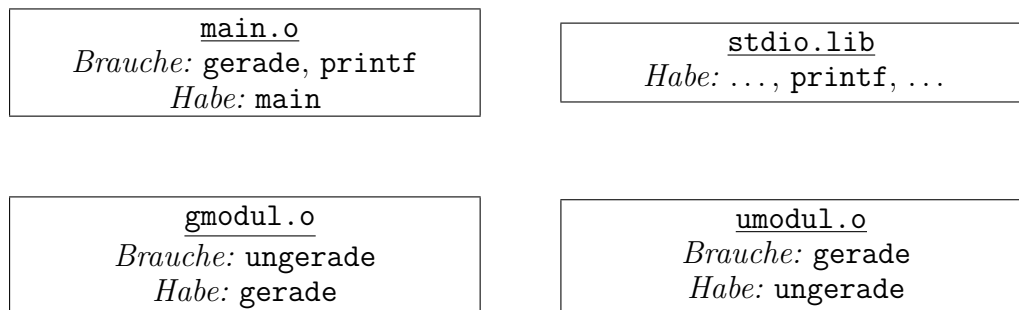
```

1 int ungerade(unsigned n);
2 int gerade(unsigned n);
3
4 int gerade(unsigned n) {
5     if (n==0) return 0;
6     else return ungerade(n-1);
7 }

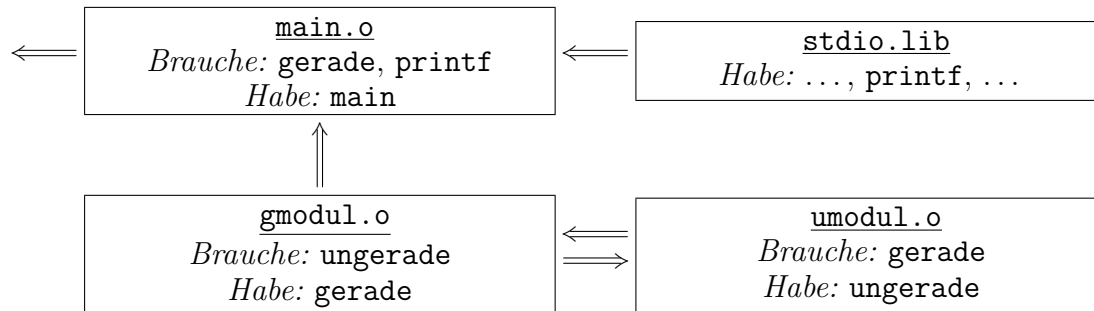
```

Listing 26: gmodul~.c

Der Compiler kompiliert diese drei Dateien nun unabhängig voneinander in Objektdateien. Dieser Vorgang kann erfolgreich durchgeführt werden, da der Compiler jede aufgerufene Funktion bereits aus einer Deklaration kennt. Wir erhalten drei Module (und haben eine Systembibliothek):



Der Linker hat nun diese Objektdateien vor sich liegen und die Aufgabe, sie miteinander zu verknüpfen. Er erstellt eine große Binärdatei und verknüpft die Funktionsaufrufe angemessen:



Der Linker muss eine Funktion mit Namen main finden. Diese wird vom fertig verlinkten Programm als Einstiegspunkt verwendet.

## 4 Adressierung und Arrays

### 4.1 Adressen und Pointer

Wie bereits bekannt, lassen sich eine oder mehrere Speicherzellen zu Variablen zusammenfassen, in denen verschiedene Datentypen gespeichert werden können. Bereits bekannt ist auch, dass die Speicherzellen sequentiell durchnummeriert sind - die Nummer der ersten Speicherzelle einer Variablen nennt man auch ihre *Adresse*. Um die Adresse einer Variablen (als Zahl) zu erhalten, verwendet man in C den sogenannten *Adressoperator* `&` :

```
1 int main () {  
2   double euler = 2.7;  
3   printf("%lu\n", (unsigned long) &euler); /* Gibt die Adresse von  
   euler aus */  
4   return 0;  
5 }
```

Listing 27: Der Adressoperator

Eine Variable, welche die Adresse einer anderen Variablen speichert, nennt man einen *Pointer*. Ein Pointer hat selbst die Größe eines CPU - Registers, damit die CPU Speicheradressen in Registern halten und gleichzeitig möglichst viel Speicher auf einmal verwalten kann. Um die Variable selbst aus einem Pointer zurückzugewinnen, verwendet man den *Dereferenzierungsoperator* `*` :

```
1 int main () {  
2   double euler = 2.7;  
3   printf("%f\n", *(&euler)); /* Gibt euler selbst aus */  
4   return 0;  
5 }
```

Listing 28: Der Dereferenzierungsoperator

Um mit Pointern als tatsächlichen Variablen in C arbeiten zu können, müssen zwei Mehrdeutigkeiten aufgelöst werden:

- Aus der Nummer einer Speicherzelle ist nicht ersichtlich, was für eine Variable an dieser Adresse im Speicher liegt - verschiedene Variablentypen unterscheiden sich durch ihre Interpretation oder belegen sogar unterschiedlich viele Speicherzellen.
- Es ist möglich, die Adresse eines Pointers abzuspeichern, also die Adresse einer Variable, die die Adresse einer anderen Variable enthält. Es ist nicht klar, ob eine Adresse auf einen weiteren Pointer oder eine nichtpointer - Variable verweist.

Diese Probleme werden in C syntaktisch so gelöst, dass jeder Expression ein sogenanntes *Dereferenzierungslevel* zugeordnet wird. Dieses bezeichnet die Anzahl der Dereferenzierungen, die mit dem Wert durchgeführt werden müssen, damit das Ergebnis kein Pointer mehr ist. Eine Variable im herkömmlichen Sinne hat somit Dereferenzierungslevel 0. Ein gewöhnlicher Pointer hat Dereferenzierungslevel 1, ein Pointer-Pointer hat Level 2, und so weiter.

Damit erweitert sich die Variablendeklaration um folgendes Detail: Wenn eine Variable Dereferenzierungslevel  $n > 0$  haben soll, so schreibt man bei der Deklaration  $n$  Sternchen vor den Variablennamen. Auch Funktionen erhalten  $n$  Sternchen vor ihrem Namen, wenn sie Variablen zurückgeben, die ein Dereferenzierungslevel  $n > 0$  haben. Wir haben jedoch bisher keine Verwendung für Funktionen, die Pointer zurückgeben: Würden sie die Adresse einer ihrer lokalen Variablen zurückgeben, so wäre diese Rückgabe buchstäblich wertlos, da diese Variablen nach Ausführung der Funktion gelöscht werden. Wir wollen uns ein sinnvolles Anwendungsbeispiel für Pointer ansehen:

```

1  /* swap() : Zwei int-Variablen vertauschen
2     Input : Die Adressen zweier Variablen a und b
3     Ergebnis : b enthält den Wert von a und umgekehrt */
4
5  void swap(int *address1, int *address2) {
6      int temporary = *address1;
7      *address1 = *address2;
8      *address2 = temporary;
9  }
10
11 int main() {
12     int a = 10, b = 7;
13     swap(&a, &b);
14     printf("%i\n", a);
15     printf("%i\n", b);
16     return 0;
17 }

```

Listing 29: Variableninhalt vertauschen

## 4.2 Statische Arrays

Ein Array sind mehrere, im Speicher direkt aufeinanderfolgende Variablen vom gleichen Typ, welche durch ihren Abstand (engl.: Offset) vom ersten Element indiziert werden. Einen Array mit ANZAHL Elementen deklariert man durch

DATENTYP ARRAYNAME[ANZAHL] = { INITIALISIERUNG } ;

wobei die Anzahl der Elemente immer eine Konstante sein muss - daher bezeichnet man solche Arrays auch als *statisch*. Die Initialisierung ist eine Expressionsliste, welche maximal so viele Einträge haben darf, wie der Array Elemente



aufnehmen kann. Hat die Expressionliste weniger Einträge, so werden alle nachfolgenden Elemente des Arrays zu 0 initialisiert:

```
1 double point[2] = { 1, 5 }; /* point wird {1.0,5.0} */  
2 int a[10] = { 1, 2, 3, 4 }; /* a wird {1,2,3,4,0,0,0,0,0,0} */
```

Um auf die einzelnen Variablen zuzugreifen, verwendet man eckige Klammern , um den Index anzugeben:

ARRAYNAME[INDEX]

Dabei kann der Index eine beliebige ganzzahlige Expression sein. Das erste Element eines Arrays hat den Index 0. In obigem Beispiel wäre etwa `a[2]` eine Variable, welche nach Initialisierung den Wert 3 hat. An dieser Stelle sei angemerkt, dass Zugriff über die Grenzen eines Arrays hinaus in C durchaus möglich ist - es bedeutet einen Zugriff auf den Speicherbereich hinter dem Array. Dies führt jedoch zu unvorhersehbarem Verhalten und meist zum Absturz des Programms. Um auf Elemente des Arrays zuzugreifen, genügt es, dessen Anfangsadresse zu kennen: Daher verhalten sich statische Arrays in C *fast* wie ein Pointer auf das erste Element des Arrays. Dieser Pointer ist jedoch nicht veränderbar – er zeigt *statisch* auf das erste Element des Arrays. In eine Arrayvariable `a` selbst darf daher nicht mit “=” direkt geschrieben werden, sondern nur in die Variablen `a[i]` für ganzzahliges `i`. Insbesondere kann man das Gleichheitszeichen *nicht* benutzen, um ein Array in ein anderes zu kopieren. Um es zu verändern, muss ein Array elementweise modifiziert werden. Zusammenfassend: Jedes Element des Arrays muss einzeln, durch indizierten Zugriff verändert werden.

Man kann nun den Wert einer Arrayvariable (die Adresse des ersten Elements) einem Pointer mit gleichem Datentyp zuweisen: Dieser zeigt dann auf das erste Element des Arrays. Wir werden im nächsten Abschnitt erfahren, welchen Zweck dies erfüllt.

### 4.3 Pointerarithmetik

C ermöglicht es, auf Pointern arithmetische Operationen durchzuführen. Dazu definieren wir zunächst die *Größe* eines Datentyps als die Anzahl der Bytes, die eine Variable diesen Typs im Speicher belegt. Die Größe eines Datentyps lässt sich durch den Compilerbefehl `sizeof` bestimmen:

`sizeof`(DATENTYP)  
`sizeof`(VARIABLENNAME)

An der Stelle eines `sizeof`-Befehls schreibt der Compiler eine Ganzzahlkonstante, welche der Anzahl Speicherzellen entspricht, die eine Variable des angegebenen Typs bzw. die angegebene Variable beansprucht. So wird etwa `sizeof(double)`

zur Konstante 8. Die Größe einer Pointervariable entspricht der Größe eines CPU-Registers (in Bytes). In C ist die Addition einer ganzen Zahl  $n$  zu einem Pointer  $p$  so definiert, dass zu der Adresse  $p$  das  $n$ -fache von  $d$  addiert wird, wobei  $d$  die Größe des durch  $p$  referenzierten Datentyps ist. Das additive Verknüpfen von Pointern mit ganzen Zahlen wird als *Pointerarithmetik* bezeichnet. In diesem Zusammenhang sind alle gängigen Operator-Kurzschreibweisen ( $++$ ,  $--$ ,  $+=$ ,  $-=$ ) weiterhin verwendbar. Ist nun `ptr` ein Pointer oder ein statisches Array und  $i$  eine Ganzzahlexpression, so werden die eckigen Klammern eines Indexzugriffs wie folgt vom Compiler übersetzt:

$$\text{ptr}[i] \triangleq \text{*(ptr + i)}$$

Diese Übersetzung findet jedes Mal statt, wenn eckige Klammern verwendet werden. Damit sind also eckige Klammern eine Kurzschreibweise für Pointerarithmetik, kombiniert mit einer Dereferenzierung.

Wenn wir, wie im vorangegangenen Abschnitt besprochen, einem Pointer die Adresse des ersten Elements eines statischen Arrays zuweisen, so können wir den Pointer danach wie das Array selbst verwenden. Dies ist nützlich, wenn Funktionen mit dem Inhalt eines Arrays arbeiten sollen. In diesem Fall übergeben wir für gewöhnlich einen Pointer auf das erste Element des Arrays, wie im folgenden Beispiel:

```

1 #include <stdio.h>
2
3 /* bestimme das maximum eines arrays */
4 double array_max(unsigned *array, unsigned length) {
5     unsigned i, max = 0;
6     for (i=0; i<length; i++)
7         if (array[i] > max) max = array[i];
8     return max;
9 }
10
11 int main() {
12     unsigned a[5] = { 12, 9, 1, 3, 7 };
13     printf("%f\n", array_max(a,5));
14 }
```

Die Übergabe eines Arrays als Pointer an Funktionen bietet Vorteile. Eine Funktion kann ein uninitialisiertes Array als Argument erhalten, um es mit Inhalt zu füllen und so eine vektorwertige Rückgabe zu liefern. Weiterhin muss nicht jedes mal das komplette Array kopiert werden, um es einer Funktion zu übergeben (lediglich der Pointer wird übergeben).

## 4.4 Zeichenketten

Eine **char**-Variable speichert kleine, ganzzahlige Werte, welche als Buchstaben interpretiert werden. Mit einfachen Anführungszeichen eingefasste, einzelne Buchstaben, sind in C daher Ganzzahlkonstanten und können einer **char**-Variable zugewiesen werden:

```
1 char l = 'B';
```

Listing 30: Deklaration und Initialisierung einer Variable vom Typ char

Allerdings entspricht dieser Buchstabe lediglich einer Zahl. Die folgende Variablendeklaration würde l ebenfalls auf den Buchstaben B setzen:

```
1 char l = 66;
```

Listing 31: Initialisieren mit einer Zahl

Auch besondere Zeichen wie etwa Zeilenumbruch oder Tabulatoren haben eine Kodierung als **char**. Die Konstanten für solche sogenannten *Steuerzeichen* werden durch einen Backslash eingeleitet:

Konstante	Beschreibung des Zeichens
'\n'	Zeilenumbruch
'\r'	Wagenrücklauf
'\t'	Tabulator
'\a'	Klingel (erzeugt ein Piepen)
'\0'	Nullcharakter (siehe unten)
'\\'	Backslash ausgeben
'\''	Hochkomma ausgeben

Tabelle 9: Steuerzeichen

Eine *Zeichenkette* (oder *String*) ist ein Array von **char**-Variablen. Man verwendet sie, um Text zu speichern und zu verarbeiten. Allerdings kann sich die tatsächliche Länge eines Textes bei der Verarbeitung häufig ändern, unter Umständen ist der Text kürzer als der Array, den man zum Abspeichern des Textes verwenden möchte. Daher verwendet man einen besonderen Wert, um das Ende eines Strings zu markieren: Die char-Variable, die nach dem letzten Buchstaben des Textes im Array kommt, erhält den Wert `'\0'`. Man bezeichnet dies auch als den *Nullcharakter* und spricht in diesem Zusammenhang von *nullterminierten Strings*. Sobald man bei der Verarbeitung von Strings auf den Nullcharakter trifft, so markiert dies das Ende des Strings, selbst wenn das Array mehr Speicherzellen enthält.

Man kann auch ganze *Zeichenketten* als Konstante angeben. Zu diesem Zweck verwendet man doppelte Anführungszeichen - Konstante Strings sind uns bereits

bei zahlreichen `printf()` Aufrufen über den Weg gelaufen. Mit einem konstanten String kann man ein `char`-Array initialisieren:

```
1 char L[100] = "Cola";
```

Listing 32: Initialisierung eines `char`-Arrays durch einen String

Dies ist lediglich eine einfachere Schreibweise für

```
1 char L[100];  
2 L[0] = 'C';  
3 L[1] = 'o';  
4 L[2] = 'l';  
5 L[3] = 'a';  
6 L[4] = '\0';
```

Listing 33: Initialisierung eines `char`-Arrays durch viele chars

*Achtung:* Der Array muss also mindestens Platz für alle Zeichen des Strings und den Nullcharakter als Abschlussmarkierung bieten, sofern man diese Initialisierung verwenden möchte. Es ist jedoch *nicht* möglich, einer Arrayvariable eine Stringkonstante später zuzuweisen, da man das Array nur elementweise verändern kann.

Als die *Länge* einer Zeichenkette bezeichnet man die Anzahl der Einträge eines `char`-Arrays vor dem abschließenden Nullcharakter. Wir werden später mehr über die Arbeit mit Zeichenketten lernen.

## 5 Dynamische Speicherverwaltung

### 5.1 Einleitung: Speicherverwaltung

Wenn ein Programm Speicher verwendet, so muss es diesen als “belegt” markieren, damit der Speicher nicht anderweitig vergeben wird: Auf einem Computer laufen noch weitere Programme, und es muss Klarheit darüber herrschen, welches davon auf welche Speicherbereiche zugreift. Wenn wir bisher Speicher verwendet haben, dann immer in Form einzelner Variablen oder statischer Arrays: Dies sind Regionen im Speicher, von denen schon zur Zeit der *Kompilierung* fest steht, wie groß sie sein werden. Daher kann der Compiler den notwendigen Code erzeugen, um sich diesen Speicher zu reservieren (und ihn anschließend auch wieder freizugeben). Es gibt jedoch Situationen, in denen wir zur Laufzeit des Programms erst berechnen, wie viel Speicher benötigt wird. In diesem Fall müssen wir Speicher manuell reservieren und wieder freigeben.

### 5.2 Allokierung von Speicher

Wenn die Länge eines im Programm benötigten Arrays nicht von vornherein bekannt ist, sondern erst während der Laufzeit berechnet werden muss, so kann man kein statisches Array verwenden. Zu diesem Zweck gibt es in `<stdlib.h>` eine Funktion zum Erstellen von Arrays dynamischer Größe. Die Anzahl der benötigten Speicherzellen wird an diese Funktion als Parameter übergeben und muss somit nicht konstant sein. Die Funktion findet dann eine unbelegte, ausreichend große und zusammenhängende Region im Speicher, markiert diese als “belegt” und liefert einen Pointer auf das erste Byte der Region als Rückgabewert. Diesen Vorgang bezeichnet man auch als *Allokierung* von Speicher. Die Deklaration dieser Funktion lautet:

```
1 void *malloc(unsigned int length);
```

Listing 34: Funktionsdeklaration malloc

Aber Vorsicht - um genug Speicherzellen für einen Array von  $n$  `double` - Variablen zu erhalten, muss man natürlich

```
1 double *array = malloc(n * sizeof(double));
```

aufrufen, da jede Variable gerade `sizeof(double)` Speicherzellen beansprucht. Das Analoge gilt natürlich auch für alle anderen Datentypen.

Um nun noch den Rückgabewert von `malloc` zu verstehen, wollen wir die Erklärung des Datentyps `void` aus 3.1 vervollständigen: Man kann zwar keine `void`-Variablen (mit Dereferenzierungslevel 0) erstellen, doch es ist möglich, Pointer auf `void` zu deklarieren. Eine solche Variable nennt man *Voidpointer*. Dieser speichert zwar eine Adresse, kann aber nicht dereferenziert werden - man kann seinen

Wert jedoch einer Pointervariable mit gleichem Dereferenzierungslevel und unterschiedlichem Typ zuweisen. Dies soll symbolisieren, dass man den Rückgabewert von `malloc` einer Pointervariable beliebigen Typs zuweisen kann, solange sie ein Dereferenzierungslevel von *mindestens* 1 hat.

### 5.3 Speichermangel

Es kann passieren, dass `malloc` nicht die angeforderte Menge Speicher allokieren kann. In diesem Fall gibt `malloc` den Wert `NULL` zurück. Dies ist ein Pointer, der eine spezielle Speicheradresse (meistens die Adresse 0) enthält. Man bezeichnet solche Pointer auch als *Nullpointer*. Damit dies nicht zu Konflikten führt, wird die Speicherzelle mit der Adresse `NULL` niemals verwendet. Jeder Pointer mit Wert `NULL` zeigt somit auf “illegalen” Speicher und kann nicht dereferenziert werden. Man sollte sich stets nach einem Aufruf von `malloc` überzeugen, keinen Nullpointer zurück bekommen zu haben. In unserem obigen Beispiel etwa so:

```
1 double *array;
2 if ( (array = malloc(n * sizeof(double))) != NULL ) {
3     /* Weiterer code: Verwende array normal */
4 } else {
5     /* Fehlerbehandlung: Kein Speicher */
6 }
```

Listing 35: Rückgabewert von `malloc` prüfen

### 5.4 Speicher freigeben

Ein “Nachteil” von dynamischen Arrays besteht darin, dass derartiger, als belegt markierter Speicher, auch vom Programmierer manuell wieder *freigegeben*, also als “nicht belegt” markiert werden muss, sobald der Array nicht mehr gebraucht wird. Dazu verwendet man die Funktion

```
1 void free(void *array);
```

Listing 36: Funktionsdeklaration `free`

Analog zur Rückgabe von `malloc` erhält `free` einen Voidpointer als Argument, so dass man beliebige Pointervariablen übergeben kann. Dieser muss die Adresse des ersten Bytes der allokierten Speicherregion enthalten. Es ist nicht nötig, die Länge des allokierten Speichers mit anzugeben - diese Information werden durch die Funktionen aus dem `malloc` Modul an anderer Stelle gespeichert. Unglücklicherweise ist im C-Standard keine Funktion vorgesehen, die diese Informationen ausliest: Man muss sich die Länge des allokierten Arrays also selbst in einer weiteren Variable merken.

Die Freigabe von Speicher ist unablässig: Wird in einem Programm immer und immer wieder Speicher allokiert, ohne ihn freizugeben, tritt auf kurz oder lang der Fall 5.3 ein. Allerdings muss man den Speicher nicht unbedingt in der Funktion wieder freigeben, in der man ihn allokiert - man gibt den Speicher frei, sobald man ihn nicht mehr braucht.

Um noch ein Beispiel zu liefern: So könnte eine Funktion zum erstellen von dynamischen `double`-Arrays aussehen:

```
1 double *double_malloc( unsigned long count ) {  
2     return malloc( count * sizeof(double) );  
3 }
```

Listing 37: Funktion zur Allokierung eines Arrays vom Typ `double`

## 5.5 Speicherbereiche verändern

Es ist möglich, die Größe eines durch `malloc` allokierten Speicherbereiches zu verändern. Dazu verwendet man die Funktion

```
1 void *realloc(void *array, unsigned int new_length);
```

Sofern `array` der Nullpointer ist, verhält sich `realloc` genau wie ein Aufruf von `malloc` mit Argument `new_length`. Ist `array` allerdings die Adresse eines bereits allokierten Arrays, so versucht die Funktion, die Größe des allokierten Speichers auf die Größe `new_length` zu bringen. Bei Erfolg bleibt der bisherige Inhalt unverändert – es sei denn, der Speicherbereich wurde verkleinert. In den folgenden zwei Fällen gibt `realloc` den Nullpointer zurück:

- `new_length` hatte den Wert 0. Dann wurde `array` erfolgreich freigegeben.
- `new_length` war größer als die ursprüngliche Größe von `array` und es ist nicht genug Speicher verfügbar. In diesem Fall bleibt der Speicher an der Stelle `array` unverändert.

Wenn nicht der Nullpointer zurückgegeben wird, so ist der Rückgabewert von `realloc` ein Pointer auf das vergrößerte Array. Dieser wird nicht unbedingt der gleiche sein, der an `realloc` übergeben wurde: Wenn hinter dem bereits allokierten Speicher nicht genug Platz ist, wird `realloc` an einer anderen Stelle genug Speicher allokiert, den Inhalt von `array` dorthin kopieren und den alten Speicher freigeben. Achtung also: Man sollte `realloc` immer wie folgt verwenden:

```

1 double *array, *tmp;
2 unsigned int n; /* Anzahl Elemente des Arrays */
3 /* ... */
4 if (tmp = realloc(array, n*2*sizeof(*array) )) {
5     array = tmp;
6     n *= 2;
7     /* Speicher verwenden */
8 } else {
9     /* Fehlerbehandlung */
10 }

```

## 5.6 Funktionen zum Speichermanagement

Neben den Allokierungsfunktionen benötigt man meistens noch einige zusätzliche Funktionen, um den Inhalt von Speicherbereichen zu manipulieren. Alle nachfolgenden Funktionen sind nicht in `<stdlib.h>` deklariert, sondern in `<string.h>`. Um den Inhalt von Speicherregionen zu kopieren, kann man die Funktion `memmove` verwenden:

```

1 void *memmove(void *dest, void *src, unsigned int count);

```

Sie kopiert die ersten `count` Bytes von `src` auf die ersten `count` Bytes bei `dest`. Gelegentlich ist es noch nützlich, alle Speicherzellen eines Speicherblocks auf einen bestimmten Wert `b` (meistens 0) setzen zu können. Dazu verwendet man die Funktion

```

1 void *memset(void *dest, int b, unsigned int count);

```

Irreführenderweise ist das zweite Argument als `int` deklariert, obwohl natürlich nur Werte im Bereich von 0 bis 255 zulässig sind: Von dem übergebenen `int` betrachtet `memset` nur das unterste Byte. Die Funktion setzt dann die ersten `count` Bytes an der Adresse `dest` auf `b`. Als letztes beschreiben wir noch eine Funktion, um den Inhalt zweier Speicherregionen zu vergleichen:

```

1 int memcmp(void *a, void *b, unsigned int count);

```

Die Funktion liefert den Wert 0, wenn der Inhalt des Speichers bei `a` (die `count` aufeinanderfolgenden Bytes bei `a`) und bei `b` der gleiche ist. Wenn sich die Speicherinhalte unterscheiden, so vergleicht `memcmp` sie lexikographisch<sup>1</sup> und gibt genau dann einen Wert kleiner/größer 0 zurück, wenn `a` kleiner/größer ist als `b`.

---

<sup>1</sup>Nach lexikographischer Ordnung ist ein Array von Bytes `a` kleiner als ein ebenso langes Array von Bytes `b`, wenn der erste Eintrag von `a`, in dem sich beide Arrays unterscheiden, kleiner ist als der entsprechende Eintrag von `b`.



## 5.7 Stringmanipulation

Speicher für Strings lässt sich freilich über `malloc` allokieren. Da Strings aber eine spezielle Art von Array sind, gibt es in `<string.h>` noch einige Funktionen speziell für das Arbeiten mit Zeichenketten. Etwa sollte man statt `memmove` für Strings eher die Funktion

```
1 char *strcpy(char *dest, char *src);
```

verwenden. Diese überschreibt den String bei `dest` mit dem bei `src`, inklusive des abschließenden Nullzeichens. Da Strings nullterminiert sind, muss auch kein Längenargument übergeben werden. Es ist natürlich Vorsicht bei der Verwendung geboten, da die Funktion `strcpy` immer davon ausgeht, dass bei `dest` genügend Speicherplatz verfügbar ist für die Zeichen aus `src` (inklusive abschließendem Nullzeichen.)

Möchte man die Länge eines Strings bestimmen, so verwendet man die Funktion

```
1 int strlen(char *str);
```

Besonders häufig möchte man zwei Strings “verketteten” - also hinter das Ende eines Strings die Zeichen eines anderen Strings kopieren. Die Verkettung von "Coca\0" mit "Cola\0" wäre dann "CocaCola\0" - die abschließende Null des ersten Strings wird also bei diesem Vorgang überschrieben. Um zwei Strings zu verketteten, verwendet man die Funktion

```
1 char *strcat(char *s1, char *s2);
```

Auch hier wird davon ausgegangen, dass hinter `s1` ausreichend Speicherplatz verfügbar ist, um alle Zeichen aus `s2` zu beherbergen. Um die Funktionsweise weiter zu verdeutlichen hier eine Möglichkeit, wie `strcat` mit den bereits bekannten Funktionen `strlen` und `strcpy` implementiert sein könnte:

```
1 char *strcat(char *s1, char *s2) {  
2     strcpy(s1+strlen(s1), s2);  
3     return s1;  
4 }
```

Um schlussendlich die Kopie eines Strings zu erstellen, verwendet man die Funktion

```
1 char *strdup(char *str);
```

Sie allokiert ausreichend Speicher um den gesamten String bei `str` inklusive abschließendem Nullzeichen zu beherbergen, kopiert den String an diese Stelle und gibt einen Pointer auf den neuen String zurück.

Eine Implementierung könnte etwa wie folgt aussehen:

```

1 char *strdup(char *str) {
2     char *copy;
3     unsigned length = (strlen(str)+1) * sizeof(char);
4     /* allokieren Platz für String und Terminierungszeichen */
5     if (copy = malloc(length)) {
6         return memmove(copy, str, length);
7     } else return NULL;
8 }

```

Der von `strdup` zurückgegebene Pointer *muss* also im weiteren Verlauf des Programms irgendwann durch Übergabe an `free` wieder freigegeben werden!  
Um zwei Strings zu vergleichen, verwendet man

```

1 int strcmp(char *a, char *b);

```

Die Rückgabe von `strcmp` ist 0 genau dann, wenn die Strings gleich sind (also gleiche Länge haben und gleiche Zeichen enthalten). Andernfalls ist die Rückgabe größer (bzw. kleiner) als 0, wenn `a` lexikographisch größer (bzw. kleiner) als `b` ist. Es gibt noch einige weitere Funktionen in der `<string.h>`, doch lassen sich diese auch in einer geeigneten C-Referenz nachschlagen.

## 6 Eingabe und Ausgabe

### 6.1 Einschub: Kommandozeilenargumente

Ein Programm, kann bei seinem Aufruf in der Kommandozeile zusätzliche Parameter als Zeichenfolgen übergeben bekommen. Um auf diese Parameter zugreifen zu können, ändern wir die Definition der `main`-Funktion zu

```
1 int main( int count, char **args );
```

Das Betriebssystem übergibt dann bei Ausführung des Programms einen Array von Strings in `args` und dessen Länge in `count`. Die Strings `args[0]` bis `args[count-1]` enthalten dann die Kommandozeilenargumente. Betrachte etwa folgendes Programm `arg_print.c`:

```
1 #include <stdio.h>
2
3 int main( int count, char **args ) {
4     int i;
5     for (i = 0; i < count; i++)
6         printf("Argument %i: %s\n",i,args[i]);
7     return 0;
8 }
```

Auf der Kommandozeile würde diese Code das folgende Ergebnis haben:

```
$ gcc -Wall -Wpedantic -o arg_print arg_print.c
$ ./arg_print Dies ist ein Test
Argument 0: ./arg_print
Argument 1: Dies
Argument 2: ist
Argument 3: ein
Argument 4: Test
$ ./arg_print "Dies ist ein Test"
Argument 0: ./arg_print
Argument 1: Dies ist ein Test
```

Es fällt auf, dass `arg[0]` stets den Dateinamen des aufgerufenen Programms enthält. Auf diese Art und Weise kann einem Programm etwa der Dateiname einer Datei übergeben werden, mit der es arbeiten soll – was uns zum eigentlichen Thema bringt.

## 6.2 Dateien öffnen

Um Dateien auszulesen oder zu beschreiben, muss man sie zunächst “öffnen”. Damit teilt man dem Betriebssystem mit, dass von diesem Zeitpunkt an keine Änderungen an der Datei vorgenommen werden sollen (etwa durch andere Programme.) Dazu benötigt man aus dem Modul `<stdio.h>` die Funktion `fopen` (für “file open”):

```
1 FILE *fopen(char *pfad, char *modus);
```

Der Pfad gibt an, wo auf der Festplatte die Datei liegt. Ein Pfad unter Windows wäre beispielsweise

`C:\Eigene Dateien\Dokumente\foo.txt`

und unter Linux etwa

`/home/rattle/dokumente/foo.txt`

Pfadangaben können auch “relativ” sein: Wenn sich in dem Ordner, in dem das Programm liegt, auch eine Datei `foo.txt` befindet, so würde die Angabe des Strings `"foo.txt"` bei einem Aufruf von `fopen` genügen.

Der Modus gibt zusätzliche Informationen darüber an, wie die Datei geöffnet werden soll. Der Modus kann eine der folgenden Zeichenketten sein:

"w"	Erstellt die Datei und öffnet sie zum Schreiben. Vorsicht: Sollte sie bereits existieren, wird sie gelöscht und eine neue mit dem gleichen Namen erstellt.
"r"	Öffnet eine bereits existierende Datei zum Lesen. Sollte die Datei nicht existieren, wird <code>fopen</code> einen Nullpointer zurückgeben.
"a"	Sofern die Datei existiert, wird sie zum Schreiben ans Dateiende geöffnet. Andernfalls wird sie erstellt und zum Schreiben geöffnet.
"w+"	Verfährt wie "w" mit dem Unterschied, dass nach dem Öffnen auch aus der Datei gelesen werden kann.
"r+"	Verfährt wie "r" mit dem Unterschied, dass nach dem Öffnen auch in die Datei geschrieben werden kann.
"a+"	Verfährt wie "a" mit dem Unterschied, dass nach dem Öffnen auch aus der Datei gelesen werden kann.

Der Rückgabewert der Funktion ist ein sogenannter *Filepointer*. Dies ist ein Pointer auf eine sogenannte “Struktur” mit Namen `FILE`, in der das Betriebssystem Informationen über die derzeit geöffnete Datei speichert. Eine Struktur ist eine Zusammenfassung mehrerer Variablen unter einem Namen - dazu später mehr. Eine dieser Variablen ist der sogenannte *Dateicursor*. Er gibt an, wo man sich

gerade beim Arbeiten in der Datei befindet. Alle nun folgenden Lese- und Schreiboperationen beginnen an dieser Stelle in der Datei und verändern diesen Wert um die Anzahl der Zeichen, die gelesen bzw. geschrieben wurden: Daher erhalten alle Lese- und Schreibfunktionen einen Pointer auf die **FILE**-Struktur. Beim Öffnen einer Datei zum Lesen oder Schreiben ist der Dateicursor anfangs gleich 0. Wird die Datei zum Schreiben ans Dateiende geöffnet, so verweist der Cursor direkt auf das Dateiende.

Sollte das Öffnen der Datei fehlschlagen, so gibt **fopen** den Nullpointer zurück. Nachdem man eine Datei erfolgreich geöffnet und aus ihr gelesen oder sie beschrieben hat, muss man sie wieder schließen, damit von nun an andere Programme wieder frei über die Datei verfügen können. Dazu verwendet man die Funktion

```
1 int fclose(FILE *fp);
```

Die Verwendung der Funktion ist intuitiv klar. Der Rückgabewert gibt an, ob die Datei geschlossen werden konnte und ist bei Erfolg gleich 0. Es treten in der Praxis jedoch beruhigender Weise so gut wie niemals Fehler beim Schließen von Dateien auf.

## 6.3 In Dateien schreiben

Um in eine zum Schreiben geöffnete Datei zu schreiben, stehen viele Funktionen zur Verfügung. Jede solche Funktion erhält unter anderem den Filepointer als Argument und verändert die dadurch referenzierte Struktur nach Beschreiben der Datei dahingehend, dass der Schreibvorgang beim nächsten Aufruf hinter den bereits geschriebenen Daten fortgesetzt wird. Die rudimentärste Funktion zum Schreiben in eine Datei ist

```
1 unsigned fwrite(  
2     void      *buffer,  
3     unsigned  size,  
4     unsigned  count,  
5     FILE      *fp  
6 );
```

Sie liest **count** Speicherblöcke der Größe **size** von der Adresse **buffer** und schreibt diese in die durch **fp** geöffnete Datei. Die Rückgabe der Funktion gibt an, wie viele Speicherblöcke erfolgreich in die Datei geschrieben worden sind. Ist diese Zahl kleiner als **count**, so ist ein Fehler aufgetreten. Dies kommt selten vor. Da diese Funktion jedoch relativ unspezifisch ist, und das Schreiben formatierter Dateien ausgesprochen aufwendig macht, verwendet man zum Erstellen von Textdateien mit gewissem Format häufig die Funktion

```
1 int fprintf(FILE *fp, char *format, ...);
```

Diese etwas seltsam anmutende Syntax wird nun im Detail erläutert. Die Funktion erhält als erstes Argument den Filepointer und als Zweites einen String, den sogenannten *Formatstring*. Dieser soll in die Datei geschrieben werden. Als weitere Argumente erhält die Funktion einige (beliebig viele) Expressions beliebigen Typs, welche als Zeichenketten formatiert und an bestimmten Stellen in den Formatstring eingefügt werden sollen.

Zu diesem Zweck müssen im Formatstring an eben diesen Stellen Platzhalter der Form

**%FORMAT**

stehen. Diese werden durch die Textdarstellung der Expressions ersetzt, welche an `fprintf` übergeben wurde. Die Reihenfolge der Platzhalter im String muss natürlich der Reihenfolge der übergebenen Expressions entsprechen. Das *Format* eines solchen Platzhalters hängt vom Datentyp der Variable ab, welche dort eingefügt werden soll, und von der gewünschten Art der Darstellung. Das letzte Zeichen des Formats ist ausschlaggebend für die Art der Darstellung und heißt daher *Formatzeichen*. Ein einfaches Beispiel:

```
1 fprintf(fp, "exp(%i) = %f.", 3, exp(3));
```

Hier ist `i` das Formatzeichen zum Einfügen einer vorzeichenbehafteten Ganzzahl und `f` dasjenige zum Einfügen einer Fließkommazahl. Für die am häufigsten benötigten Formatzeichen konsultiere man die nun nachfolgende Tabelle:

Datentyp	Darstellungsform	Zeichen
<code>signed int</code>	Ausgabe als vorzeichenbehaftete Ganzzahl (Verhält sich wie <code>d</code> ).	<code>i</code>
<code>signed int</code>	Ausgabe als vorzeichenbehaftete Dezimalzahl.	<code>d</code>
<code>unsigned int</code>	Ausgabe als natürliche Ganzzahl	<code>u</code>
<code>unsigned int</code>	Ausgabe als Darstellung zur Basis 16	<code>X</code>
<code>unsigned int</code>	Ausgabe als Darstellung zur Basis 8	<code>o</code>
<code>float</code>	Ausgabe als Kommazahl	<code>f</code>
<code>float</code>	Ausgabe in wissenschaftlicher Notation	<code>e</code>
<code>char</code>	Ausgabe des kodierten Zeichens	<code>c</code>
<code>char*</code>	Einfügen des referenzierten Strings	<code>s</code>
<code>void*</code>	Ausgabe der Adresse, zur Basis 16	<code>p</code>

Verwendet man die “große” Variante eines ganzzahligen Datentyps (also ein `long`), so sollte der Längenmodifikator `l` direkt vor das Formatzeichen gesetzt werden. Bei der “kurzen” Variante eines ganzzahligen Datentyps (also bei Verwendung des Ausdrucks `short`) sollte vor dem Formatzeichen ein `h` stehen. Ist die Größe des ganzzahligen Datentyps nicht explizit angegeben, so kann der Längenmodifikator weggelassen werden. Die exakte Form eines Formatausdrucks ist:

% + - 0 WEITE .PRÄZISION <sup>l</sup><sub>h</sub> FORMATZEICHEN

wobei übereinander geschriebene Elemente nicht gleichzeitig angegeben werden können. Wir wollen nun das Format im Detail und anhand von Beispielen erläutern. Für die Beispiele merken wir noch an, dass die Funktion

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

exakt nach dem selben Schema arbeitet, wobei die formatierte Ausgabe hier nicht in eine Datei geschrieben wird, sondern auf der Konsole ausgegeben wird.

**Weite** Dies muss eine ganze Zahl sein, welche die minimale Anzahl Zeichen angibt, welche für die Textdarstellung der entsprechenden Expression verwendet werden sollen. Einige Beispiele:

```
1 printf("Test: %3i",4);           /* Ausgabe: Test:   4   */
2 printf("Test: %3i",10050);       /* Ausgabe: Test: 10050 */
```

**Flags** Nach dem Prozentzeichen können beliebig viele der Flagzeichen geschrieben werden.

Flag	Effekt
-	Innerhalb der angegebenen Weite <i>links</i> ausrichten. Für gewöhnlich wird rechts ausgerichtet (siehe oben).
+	Bei vorzeichenbehafteten Zahlen <i>immer</i> das Vorzeichen mit ausgeben. Für gewöhnlich würde das Vorzeichen nur bei negativen Zahlen angezeigt.
0	Mit Nullen anstatt mit Leerzeichen auffüllen, wenn die Ausgabe zu klein für die angegebene Weite ist. Wenn 0 und - zusammen angegeben werden, wird 0 ignoriert.

**Präzision** Dies muss ebenfalls eine Ganzzahl sein (man beachte den führenden Punkt), welche nur für Ausgabe von Fließkommazahlen einen Effekt hat: Sie definiert die Anzahl der Nachkommastellen, die ausgegeben werden sollen. Beispiele:

```
1 #include <math.h>
2 /* ... */
3 printf("Test: %.3f.", sqrt(2));   /* Ausgabe: Test: 1.414. */
4 printf("Test: %.0f.", sqrt(2));   /* Ausgabe: Test: 1.     */
```

*Achtung:* Die Weite bezieht sich auf alle Zeichen der Stringdarstellung, inklusive Punkt!

```

1 printf("[%6.3f]", sqrt(2)); /* Ausgabe: [ 1.414] */
2 printf("[%7.3f]", sqrt(2)); /* Ausgabe: [ 1.414] */
3 printf("[%7.3f]", 2.3); /* Ausgabe: [ 2.300] */
4 printf("[%7.9f]", 2.3); /* Ausgabe: [2.300000000] */

```

**Längenmodifikator** Der Längenmodifikator ist nur für ganzzahlige Argumente relevant, wie oben erläutert. Wenn man ein `double` ausgibt, so verwendet man das Formatzeichen `f` ohne Längenmodifikator. Die `double`-Expression wird dann in ein `float` gecastet und als ein solches ausgegeben.

Bei Angabe der Modifikatoren wird vorne mit Leerzeichen und hinten mit Nullen aufgefüllt, sofern nötig. Wenn man gerne vorne ebenfalls mit Nullen auffüllen möchte, kann man das Flag `0` verwenden (die `0` ist das Flag, die `5` gibt die Weite an):

```

1 printf("[%05i]", 100) /* Ausgabe wird sein: [00100] */

```

Die Funktionen `printf` und `fprintf` geben die Anzahl der Zeichen zurück, die erfolgreich ausgegeben bzw. in die Datei geschrieben worden sind. Dies sind je nach Formatierung der Variablen mehr (oder sogar weniger) Zeichen als die Länge des Formatstrings.

## 6.4 Dateien lesen

Um aus einer zum Lesen geöffneten Datei zu lesen, stehen auch viele Funktionen zur Verfügung. Jede solche Funktion erhält unter anderem den Filepointer als Argument. Wenn eine Anzahl  $n$  an Bytes aus der Datei gelesen wurde, so verändert die Lesefunktion die Struktur, auf die der Filepointer verweist, dahingehend, dass der nächste Lesevorgang  $n$  Bytes später durchgeführt wird. Die rudimentärste solche Funktion ist

```

1 unsigned fread(
2     void      *buffer,
3     unsigned   size,
4     unsigned   count,
5     FILE      *fp
6 );

```

Sie liest `count` Speicherblöcke der Größe `size` aus der Datei `fp` aus und speichert diese an die Stelle im Speicher, auf die `buffer` zeigt. Die Rückgabe der Funktion gibt an, wie viele Speicherblöcke erfolgreich aus der Datei gelesen und in den Speicher kopiert worden sind. Ist diese Zahl kleiner als `count`, so wurde das Dateiende frühzeitig erreicht. Man kann auf diese Weise überprüfen, wann man den gesamten Inhalt einer Datei ausgelesen hat, indem man den Rückgabewert der



Funktion mit dem übergebenen Wert `count` vergleicht. Da diese Funktion jedoch relativ unspezifisch ist, und das Auslesen formatierter Dateien ausgesprochen aufwendig macht, verwendet man im Fall von Textdateien mit gewissem Format häufig die Funktion

```
1 int fscanf(FILE *fp, char *format, ...);
```

Diese Funktion verhält sich nahezu analog zu `fprintf` (siehe 6.3): Aus der Datei werden Zeichen gelesen, die exakt dem durch Formatstring `format` angegebenen Format entsprechen müssen. Auch in diesem Formatstring tauchen Formatausdrücke der Form `%FORMAT` auf. Zeichen des Formatstrings, die kein Formatausdruck sind, werden von `fscanf` einfach überlesen. Wenn im Formatstring ein Zeichen angegeben ist, welches an dieser Stelle nicht in der Datei steht, so bricht `fscanf` an dieser Stelle den Vorgang ab. An den Stellen, an denen im Formatstring Formatausdrücke auftauchen, erwartet `fscanf` in der Datei entsprechend formatierte Zeichenketten, welche dann in C-Expressions übersetzt werden. Die weiteren Argumente von `fscanf` sind Pointer auf Variablen, welchen diese Expressions in entsprechender Reihenfolge zugewiesen werden. Der Rückgabewert von `fscanf` gibt die Anzahl der *erfolgreich* zugewiesenen Werte zurück – oder die Konstante `EOF`, falls ein Fehler auftritt. Dies ist der Fall, wenn etwa das Ende der Datei zu früh erreicht wurde oder ein Zeichen im Formatstring vorkommt, welches in der Datei nicht an der entsprechenden Stelle steht. Noch einige Bemerkungen zum Umgang von `fscanf` mit Whitespace<sup>2</sup>:

- Falls ein Whitespace-Zeichen im Formatstring vorkommt, so darf an der entsprechenden Stelle in der Datei eine beliebige Anzahl beliebiger, aufeinanderfolgender Whitespace-Zeichen stehen – insbesondere auch gar keines.
- Falls in der Datei ein Whitespace-Zeichen vorkommt, ohne dass im Formatstring an der entsprechenden Stelle ein Formatausdruck oder Whitespace angegeben ist, so bricht die Funktion frühzeitig ab.
- Beim Einlesen eines Formatausdrucks werden führende Whitespace-Zeichen von `fscanf` einfach überlesen.

Wir wollen nun das Format noch vollständig erklären. Die Formatausdrücke im Formatstring sind bei `fscanf` von der Form:

$$\% * \text{WEITE} \frac{1}{h} \text{FORMATZEICHEN}$$

**Weite** Dies gibt die maximale Anzahl Zeichen an, die `fscanf` zum Einlesen dieses Feldes verwendet:

---

<sup>2</sup>Whitespace sind Leerzeichen, Tabulatoren und Zeilenumbrüche.

```

1 int year;
2 printf("Bitte Jahreszahl (vierstellig) eingeben: ");
3 scanf("%4i", &year); /* lies die nächsten 4 Ziffern */

```

Zum Einlesen von Variablen ist es nicht erforderlich, die Weite anzugeben. Für gewöhnlich liest **fscanf** so viele Zeichen ein, wie für einen Ausdruck des angegebenen Typs sinnvoll möglich sind.

**Flags** Es gibt nur das Sternchen **\*** als Flag für **fscanf**-Formate. Es bedeutet, dass an dieser Stelle zwar ein Ausdruck im angegebenen Format ausgelesen werden soll, dieser jedoch nirgendwo gespeichert wird. Zu einem so markierten Formatausdruck darf selbstverständlich kein Pointer als weiteres Argument an **fscanf** übergeben werden.

**Längenmodifikator** Der Längenmodifikator muss für ganzzahlige *und* Fließkommatentypen angegeben werden: Die Typen **long** und **double** erfordern den Modifikator **l**, der Typ **short** erfordert den Modifikator **h**.

**Formatzeichen** Die Formatzeichen sind die gleichen wie bei **fprintf**. Darüber hinaus ermöglicht **fscanf** noch einige weitere Optionen zum Einlesen von Strings, auf welche wir hier nicht eingehen können.

*Achtung:* Beim Einlesen von Ganzzahlen mittels **%i** ermittelt **fscanf** die Art der Zahldarstellung nach den gleichen Regeln wie in 2.3. Insbesondere wird eine führende 0 so interpretiert, dass eine Zahl im Oktalsystem vorliegt. Verwendet man das Format **%d**, so wird **fscanf** angewiesen, die Zahl auf jeden Fall im Dezimalsystem einzulesen.

Wir wollen nun ein Beispiel für die Verwendung von **fscanf** und **printf** geben. Sei eine Datei mit folgendem Inhalt gegeben:

```

1      exp(-5) = 0.01
2      exp(-4) = 0.0183156
3      exp(-3) = 0.0497
4      exp(-2) = 0.135335
5      exp(-1) = 0.3678794412
6      exp(0) = 1
7      exp(1) = 2.7182818285
8      exp(2) = 7.389
9      exp(3) = 20.08
10     exp(4) = 54.5
11     exp(5) = 148.41315
12     exp(6) = 400.0

```

Dann könnte man mit folgendem Programm den Fehler in jedem Funktionswert ermitteln (natürlich nur im Rahmen der Maschinengenauigkeit):

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main ( int count, char **args ) {
5     double d;
6     signed i;
7     FILE *f;
8     if ( count < 1 ) {
9         printf("Bitte Dateinamen angeben.\n");
10        return 1;
11    }
12    if ( (f = fopen(args[1], "r")) != NULL ) {
13        while (fscanf(f, "exp(%i) =%lf\n", &i, &d) == 2)
14            printf("Fehler bei exp(%2i) ist %le\n", i, d-exp(i));
15        fclose(f);
16        return 0;
17    } else {
18        printf("Konnte Datei nicht öffnen\n");
19        return 1;
20    }
21 }

```

Das Programm erwartet als ersten Kommandozeilenparameter den Namen der oben angegebenen Datei. Es fällt auf, dass sich die Funktionswerte nach dem Gleichheitszeichen auch einlesen lassen, obwohl im Formatstring danach kein Leerzeichen steht. Das liegt daran, dass führende Leerzeichen mit zur formatierten Darstellung einer Zahl gehören können und somit von `fscanf` gefressen werden, bis tatsächlich Ziffern folgen.

Um zu überprüfen, ob die letzte Leseoperation das Ende der Datei erreicht hat, kann man die folgende Funktion verwenden:

```

1 int feof(FILE *fp);

```

Sie gibt einen Wert ungleich 0 zurück, falls beim letzten Lesen das Ende der Datei erreicht wurde und 0, falls noch weitere Daten aus der Datei gelesen werden können.

## 6.5 Byteweiser Dateizugriff

Manchmal möchte man nur ein einzelnes Byte in eine Datei schreiben oder ein einzelnes Byte aus einer Datei auslesen. Dazu gibt es die Funktionen

```

1 int fgetc(FILE *fp);

```

und

```

1 int fputc(int c, FILE *fp);

```

wobei die Namen jeweils für “file get char” und “file put char” stehen. Von dem an `fputc` übergebene `int` wird wie bei `memset` (siehe 5.6) natürlich nur das unterste Byte verwendet. Als Rückgabe liefert `fputc` die Anzahl der geschriebenen Bytes - also 1 bei Erfolg und 0 bei einem Fehler. Die Funktion `fgetc` gibt entweder das ausgelesene Byte zurück (als `int`) oder den Fehlerwert `EOF`. Um die Funktionsweise der bereits vorgestellten Funktionen zu Verdeutlichen zeigen wir an dieser Stelle, wie diese beiden Funktionen implementiert sein könnten:

```
1 int fgetc(FILE *fp) {  
2     char c;  
3     if (fscanf(fp, "%c", &c) == 1) return c;  
4     else return EOF;  
5 }
```

```
1 int fputc(int c, FILE *fp) {  
2     return fprintf(fp, "%c", c);  
3 }
```

Hier ist eine weitere, dazu äquivalente Implementierung unter Verwendung von `fread` bzw. `fwrite`:

```
1 int fgetc(FILE *fp) {  
2     char c;  
3     if (fread(&c, 1, 1, fp) == 1) return c;  
4     else return EOF;  
5 }
```

```
1 int fputc(int c, FILE *fp) {  
2     return fwrite(&c, 1, 1, fp);  
3 }
```

## 6.6 Den Dateicursor verändern

Der Dateicursor ist ein Zahlenwert, der in der `FILE` Struktur einer geöffneten Datei gespeichert wird. Er gibt an, wie weit man sich vom Anfang der Datei befindet - Führt man eine Schreib- oder Leseoperation auf der geöffneten Datei aus, so werden die Bytes immer von dieser Stelle an gelesen bzw. geschrieben. Wie bereits angedeutet verändern Schreib- und Leseoperationen gleichermaßen den Dateicursor. Man kann den Dateicursor jedoch auch durch folgende Funktion verändern:

```
1 int fseek(FILE *fp, int k, int origin);
```

Die Funktion setzt den Dateicursor auf `k`, wenn man für `origin` die Konstante `SEEK_SET` verwendet. In diesem Fall darf `k` natürlich keine negativen Werte annehmen. Verwendet man für `origin` die Konstante `SEEK_CUR`, so wird `k` zum Dateicursor addiert. In diesem Fall kann `k` auch negativ sein, wir verschieben den Cursor bildlich gesprochen in der Datei zurück. Schlussendlich kann für `origin` auch die Konstante `SEEK_END` angegeben werden, um den Cursor relativ zum Ende der Datei zu setzen. Dafür sind selbstverständlich nur negative Werte von `k` zulässig. Wenn der Dateicursor erfolgreich verändert wurde, gibt die Funktion 0 zurück, andernfalls nicht.

Möchte man den Wert des Dateicursors auslesen, so kann man die Funktion

```
1 int ftell(FILE *fp);
```

verwenden, welche den Wert des Dateicursors zurückliefert. Damit lässt sich nun unter anderem die Länge einer Datei wie folgt bestimmen:

```
1 int flength(FILE *fp) {  
2     int temp,length;  
3     temp = ftell(fp);  
4     if (!fseek(fp, 0, SEEK_END)) {  
5         length = ftell(fp);  
6         fseek(fp, temp, SEEK_SET);  
7     }  
8     else length = -1; /* Fehlerwert */  
9     return length;  
10 }
```

## 7 Datenstrukturen

### 7.1 Strukturdefinitionen

Strukturen werden verwendet, um mehrere, im Speicher hintereinander angeordnete Variablen zusammenzufassen. Mit folgender Syntax definiert man eine Strukturtyp und damit einen ganz *neuen Datentyp*:

```
struct STRUKTURNAME {  
    DATENTYP 1 FELDDNAME 1;  
    DATENTYP 2 FELDDNAME 2;  
    ...  
    DATENTYP n FELDDNAME n;  
};
```

Eine solche Strukturdefinition steht außerhalb von Funktionscode meist am Anfang des Quellcodes (bzw. in der Headerdatei eines Moduls). Man kann im folgenden Code nun `struct STRUKTURNAME` als Datentyp verwenden – Variablen dieses Typs heißen *Strukturen* und enthalten mehrere “Untervariablen”, welche auch als *Felder* bezeichnet werden. Um auf die Felder einer Struktur zuzugreifen, verwendet man den *Punktoperator* :

STRUKTURNAME.FELDDNAME

Enthält eine Struktur X ein Feld von Typ T mit Namen y, so ist X.y die in X enthaltene Variable vom Typ T. Ein Beispiel:

```
1 #include <stdio.h>  
2  
3 /* Wir definieren eine Struktur: */  
4 struct DRESULT {  
5     unsigned Q;  
6     unsigned R;  
7 };  
8  
9 /* Nun ist struct DRESULT ein neuer Datentyp und als  
10 Rückgabewert der folgenden Funktion zulässig: */  
11 struct DRESULT divmod( unsigned a, unsigned b ) {  
12     struct DRESULT division;  
13     division.Q = a / b;  
14     division.R = a % b;  
15     return division;  
16 }  
17  
18 sint main() {  
19     struct DRESULT D = divmod(53,11);  
20     printf("53 / 11 = %u Rest %u\n", D.Q, D.R );  
21     return 0;  
22 }
```

Wir geben noch ein weiteres Beispiel:

```
1 #include <math.h>
2 #include <stdio.h>
3
4 struct PUNKT {
5     double x;
6     double y;
7 };
8
9 double length(struct PUNKT p) {
10     return sqrt( p.x*p.x + p.y*p.y );
11 }
12
13 int main() {
14     struct PUNKT p;
15     p.x = 6;
16     p.y = 7;
17     printf("Abstand des Punktes zu 0: %.5f", length(p));
18     return 0;
19 }
```

Es sei noch einmal darauf hingewiesen, dass durch die Strukturdefinition ein neuer Datentyp entsteht. Wir könnten in unserem Beispiel ebenso gut ein statisches Array mit Einträgen vom Typ `struct PUNKT` erstellen, oder Funktionen mit Rückgabewert `struct PUNKT` definieren. Strukturvariablen lassen sich auch initialisieren. Die komplette Syntax einer Strukturvariablen-Deklaration hat die Form

```
struct STRUKTURNAME VARIABLENNAME = { INITIALISIERUNG };
```

hierbei ist die Initialisierung eine Expressionliste, welche höchstens so viele Einträge haben darf, wie die Struktur Felder aufweist. In der Reihenfolge, in der die Felder in der Strukturdefinition auftreten, werden sie dann durch die Einträge der Expressionliste initialisiert. Der Rest wird, wie bei Arrays, zu 0 initialisiert. In unserem Beispiel könnten wir also ebenso gut einfach schreiben:

```
1 struct PUNKT p = { 6, 7 };
```

*Anmerkung:* Namen für Strukturen können nach den gleichen Regeln gewählt werden wie alle anderen Namen in C. Wir schreiben Strukturnamen konventionsmäßig in Großbuchstaben, um sie noch deutlicher von den primitiven Datentypen (`int`, `double`, `char`, ...) abzugrenzen.



## 7.2 Datenstrukturen

Der Begriff “Datenstruktur” bezieht sich nicht allein auf die Definition einer Struktur im obigen Sinne. Um neue Datentypen zu erstellen, braucht man auch Operatoren, um auf diesen Datentypen zu arbeiten. Ein Beispiel hier:

```
1 #ifndef _COMPLEX_H
2 #define _COMPLEX_H
3
4 /* Der neue Datentyp struct COMPLEX repräsentiert eine komplexe
   Zahl, indem wir Real- und Imaginärteil separat abspeichern. */
5 struct COMPLEX {
6     double real;
7     double imag;
8 };
9
10 /* multipliziere zwei komplexe Zahlen: */
11 struct COMPLEX mul( struct COMPLEX alpha, struct COMPLEX beta );
12
13 /* addiere zwei komplexe Zahlen: */
14 struct COMPLEX add( struct COMPLEX alpha, struct COMPLEX beta );
15
16 /* dividiere zwei komplexe Zahlen durcheinander: */
17 struct COMPLEX div( struct COMPLEX alpha, struct COMPLEX beta );
18
19 /* bilde das negative einer komplexen Zahl: */
20 struct COMPLEX neg( struct COMPLEX alpha );
21
22 /* potenziere eine komplexe Zahl mit einer ganzen Zahl: */
23 struct COMPLEX pot( struct COMPLEX alpha, int n );
24
25 #endif
```

Listing 38: Die komplexen Zahlen mit Rechengesetzen (complex.h)

```
1 #include "complex.h"
2
3 /* wir benötigen die Strukturdefinition von struct COMPLEX. Dies ist
   ein weiterer Grund, warum die Quellcodedatei ihre zugehörige
   Headerdatei für gewöhnlich einbindet. */
4
5 struct COMPLEX mul ( struct COMPLEX alpha, struct COMPLEX beta ) {
6     struct COMPLEX gamma;
7     gamma.real = alpha.real * beta.real - alpha.imag * beta.imag;
8     gamma.imag = alpha.real * beta.imag + alpha.imag * beta.real;
9     return gamma;
10 }
11
12 /* ... weitere Definitionen */
```

Listing 39: Die komplexen Zahlen mit Rechengesetzen (complex.c)

Man implementiert solche Datenstrukturen mit ihren Operatoren nun zusammen in ein Modul, damit die Verwendung des neuen Datentyps in vielen verschiedenen Programmen möglich wird. Ein Programm könnte das Modul nun in etwa so verwenden:

```
1 #include <stdio.h>
2 #include "complex.h"
3
4 int main() {
5     struct COMPLEX c = {0.6,-0.8}, d = c;
6     /* Berechne das Quadrat des Betrags von c: */
7     d.imag = -d.imag;
8     d = multiply(d,c);
9     printf("%5.3lf%+5.3lfi\n",d.real, d.imag);
10    return 0;
11 }
```

Listing 40: Anwendung komplexer Zahlen

Man sollte sich zum Zeitpunkt der Programmierung einer Datenstruktur vollkommen klar darüber sein, wie ein Programm später mit dieser Datenstruktur arbeiten soll. Manchmal ist es sogar am günstigsten zuerst ein kleines Programm zu schreiben, das die Datenstruktur verwendet (welches man ohnehin braucht, um sie zu testen) und dann erst mit Programmierung der Datenstruktur selbst zu beginnen.

### 7.3 Pointer auf Strukturen

Genau wie Arrays können Strukturen unter Umständen sehr viel Speicherplatz beanspruchen. Häufig kann man sich dann darauf beschränken, nur einen Pointer zu übergeben. Möchte man nun auf ein Feld *y* einer Struktur zugreifen, die durch einen Pointer *p* gegeben ist, so muss man einfach den Pointer dereferenzieren und auf die dadurch zurückgewonnene Strukturvariable durch den Punktoperator zugreifen:

`(*p).y`

Der Punktoperator bindet allerdings stärker als der Dereferenzierungsoperator (man sagt auch, der Punktoperator hat eine höhere *Operatorpräzedenz*, genau wie man auch sagt “Punkt- vor Strichrechnung”). Das bedeutet, dass die gerade gesetzten Klammern notwendig sind. Die Schreibweise `*p.y` entspräche `*(p.y)`, was, da der Punktoperator nicht auf Pointer definiert ist, syntaktisch falsch wäre. Da dies etwas umständlich ist, wurde in C dafür die abkürzende Schreibweise `p->y` eingeführt. Um unser Beispiel zu erweitern:

```

1 double Im(struct COMPLEX *c) { return c->imag; }
2 double Re(struct COMPLEX *c) { return c->real; }

```

Listing 41: Real- bzw. Imaginärteil einer komplexen Zahl zurück zu geben

Man bezeichnet diesen Operator auch als den *Pfeiloperator*.

## 7.4 Intermezzo: Typdefinitionen

In C ist es nicht nur möglich, sich neue Datentypen auf die oben genannte Weise zu definieren, man kann auch jedem bestehenden Datentyp einen weiteren Namen zuweisen. Dies ist möglich durch die Verwendung von **typedef**:

```
typedef SCHABLONE;
```

Die SCHABLONE hat die Syntax einer Variablendeklaration (ohne Initialisierung). Der Name der Variablen, den wir bei dieser “Deklaration” angeben ist der so entstandene Datentyp. Wird er vom Compiler in einer Variablendeklaration gefunden so wird er in der SCHABLONE durch den deklarierten Variablennamen ersetzt und an dieser Stelle eingefügt. Hier ein Beispiel:

```

1 typedef double POINT[2];
2 POINT p = { 2, 4 };

```

Der definierte Datentyp heißt `POINT`. Bei der Variablendeklaration wird der Name des Typs in der Schablone durch den Namen der deklarierten Variable (hier `p`) ersetzt. Damit ist obiger Code äquivalent zu

```
1 double p[2] = { 2, 4 };
```

Es ist gegebenenfalls wichtig, sich klarzumachen, dass **typedef** sich bei Deklaration mehrerer Variablen in einem Statement freundlich verhält:

```
1 POINT p = { 2, 4 }, q = { 1, 7 };
```

wird zu

```

1 double q[2] = { 1, 7 };
2 double p[2] = { 2, 4 };

```

Zunächst werden also alle Kommata expandiert und einzelne Variablendeklarationen aus dem Statement gemacht und *danach* die Schablone angewandt. Beachte:

<code>typedef char *STRING;</code>	$\xrightarrow{\text{wird zu}}$	<code>char *s1;</code>
<code>STRING s1, s2;</code>		<code>char *s2;</code>

Während andererseits:

$$\text{char *s1,s2;} \quad \xrightarrow{\text{wird zu}} \quad \begin{array}{l} \text{char *s1;} \\ \text{char s2;} \end{array}$$

Besonders im Zusammenhang mit Strukturdefinitionen stellen Typdefinitionen eine ernsthafte Erleichterung dar. Man betrachte etwa das wie folgt modifizierte Beispiel 38:

```
1 #ifndef _COMPLEX_H
2 #define _COMPLEX_H
3
4 struct COMPLEX {
5     double real;
6     double imag;
7 };
8
9 typedef struct COMPLEX COMPLEX;
10
11 COMPLEX mul(COMPLEX alpha, COMPLEX beta);
12 COMPLEX add(COMPLEX alpha, COMPLEX beta);
13 COMPLEX div(COMPLEX alpha, COMPLEX beta);
14 COMPLEX neg(COMPLEX alpha );
15 COMPLEX pot(COMPLEX alpha, int n );
16
17 #endif
```

Man kann Typdefinitionen mit Strukturdefinitionen direkt verbinden, indem man schreibt

```
typedef struct {
    DATENTYP 1 FELDNAME 1
    DATENTYP 2 FELDNAME 2
    ...
    DATENTYP n FELDNAME n
} NAME;
```

Danach kann man dann neue Variablen vom Typ NAME deklarieren, welche der Struktur entsprechen. Beispiel:

```
1 typedef struct {
2     double real;
3     double imag;
4 } COMPLEX;
```

## 7.5 Anwendung: Verkettete Listen

Wir wollen nun das Erlernte verwenden, um eine bekannte und wichtige Datenstruktur in C zu implementieren. Als Datenstruktur bezeichnen wir ein Schema, nach dem Daten einer gewissen Form im Computer gespeichert werden. Dazu gehören auch eine Reihe von Operationen auf der Datenstruktur, um diese zu modifizieren. Ein nahe liegendes Schema, um Daten im Computer zu Speichern, ist uns bereits bekannt: Arrays. Auch werden alle notwendigen Operationen auf Arrays von der Sprache C bereits in Form von elementarer Pointerarithmetik zur Verfügung gestellt.

Es gibt jedoch ein weiteres Schema, um einen sortierten Satz von mehreren Variablen zu speichern. Dieses Schema wird als (doppelt) verkettete Liste bezeichnet. In einer verketteten Liste wird jede Variable  $x_i$  in einen sogenannten Knoten (engl. "node") eingebettet, der neben  $x_i$  auch einen Rückwärtspointer  $p_i$  (previous) auf  $x_{i-1}$  und einen Nachfolgerpointer  $\text{next}_i$  (next) auf  $x_{i+1}$  enthält. Der erste Rückwärtspointer und der letzte Nachfolgerpointer sollen der Nullpointer sein. Diese Knoten modellieren wir als Strukturen:

```
1 typedef struct NODE {  
2     struct NODE *next;  
3     struct NODE *prev;  
4     double      data;  
5 } NODE;
```

Wir können also in der Definition der Struktur bereits Pointer auf die Struktur verwenden, die gerade im Begriff ist, definiert zu werden<sup>3</sup>. Jeder Knoten ist nun eine Struktur, die einen Pointer auf den nächsten Knoten und die gespeicherte Variable enthält. Die Liste selbst repräsentieren wir auch durch eine Struktur, welche Pointer auf das erste und letzte Element speichert:

```
1 typedef struct {  
2     NODE *first;  
3     NODE *last;  
4 } LIST;
```

Eine Liste soll (graphisch dargestellt) immer folgenden Aufbau haben:

---

<sup>3</sup>Dies liegt daran, dass Pointer ohnehin immer die gleiche Größe haben und lediglich Adressen speichern.

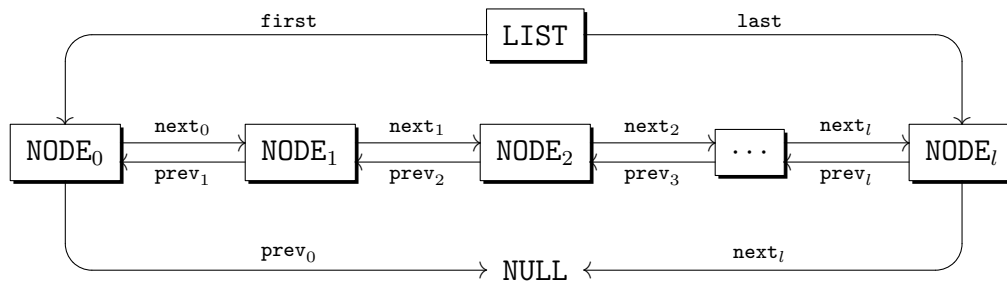
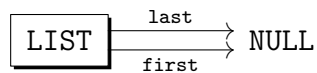


Abbildung 4: Darstellung einer doppelt verketteten Liste

Wir erlauben weiterhin, dass sowohl **first** als auch **last** der Nullpointer sind und wollen in diesem Falle sagen, die Liste sei leer:



Nun haben wir die Datentypen definiert, doch wir benötigen noch Funktionen, um mit ihnen sinnvoll arbeiten zu können. Die gesamte Headerdatei für das Modul, dass eine Listenstruktur bereitstellt, könnte etwa wie folgt aussehen:

```

1 #ifndef LIST_H
2 #define LIST_H
3
4 typedef struct NODE {
5     struct NODE *next;
6     struct NODE *prev;
7     double      data;
8 } NODE;
9
10 typedef struct {
11     NODE *first;
12     NODE *last;
13 } LIST;
14
15 LIST *list_create();          /* Liste erstellen */
16 void list_free(LIST *L);     /* Speicher freigeben */
17
18 /* Füge hinter cursor einen neuen Knoten in die Liste ein. Falls
19    cursor gleich NULL ist, füge am Anfang der Liste ein. Gibt
20    einen Pointer auf das neu eingefügte Element zurück, oder NULL
21    im Fehlerfall. */
22 NODE *list_insert(LIST *L, NODE *cursor, double data);
23
24 /* Lösche einen Knoten aus der Liste. */
25 void list_delete(LIST *L, NODE *del);
26
27 #endif

```

Es ist eine hilfreiche Übung, dieses Modul selbst zu implementieren. Wir wollen zum Schluss noch zeigen, wie man alle Elemente einer Liste fachgerecht mit einer Schleife durchläuft:

```
1 LIST *L;  
2 NODE *p;  
3 /* Liste initialisieren, Einträge erzeugen, etc. */  
4  
5 for (p=L->first; p; p = p->next) {  
6     /* arbeite mit Knoten p */  
7 }
```

Der Pointer **p** zeigt zu Beginn der **for**-Schleife auf das erste Element der Liste und springt in jedem Schritt zum nächsten. Sobald man am Ende angekommen ist, wird **p** der Wert **NULL** zugewiesen, da der **next**-Pointer des letzten Elements dorthin verweist: Die Schleife bricht ab.

## 8 Weitere Sprachkonstrukte

### 8.1 Bedingte Auswertung

Es gibt in C einen einzigen eingebauten *ternären* Operator, also einen Operator mit drei Argumenten: Die bedingte Auswertung. Er hat die Syntax

BEDINGUNG ? EXPRESSION1 : EXPRESSION2

Der Wert dieser Expression ist gleich dem Wert von EXPRESSION1, falls BEDINGUNG nicht zu 0 auswertet und andernfalls gleich dem Wert von EXPRESSION2. Als Beispiel eine recht unübersichtlich kurze Signumsfunktion:

```
1 int sgn(int a) { /* Signumsfunktion */  
2     return (a<0) ? -1 : (a?1:0);  
3 }
```

### 8.2 Konstante Variablen

Es gibt in C die Möglichkeit, manche Variablen als “konstant” zu markieren. Dies heißt, dass der Wert dieser Variablen nicht verändert werden *soll*. Dazu schreibt man den Modifikator **const** vor den Variablennamen:

```
1 double const pi = 3.141592;
```

Genauer: Wenn eine konstante Variable durch eine Zuweisung verändert wird, so gibt der Compiler eine Warnung aus – aber keinen Fehler. Wenn man also unhöflicherweise die Warnung des Compilers ignoriert, so ist es trotzdem möglich, eine konstante Variable zu verändern. Der Modifikator **const** ist demnach eine Programmierhilfe, um sicherzustellen, dass eine Variable ihren Wert nicht ändert.

Wir wollen **const** nun etwas allgemeiner verstehen: Wenn eine Variable *v* vom Dereferenzierungslevel *n* erstellt wird, so können wir hinter das *k*-te Sternchen bei der Variablendeklaration ein **const** schreiben, damit die  $(n - k)$ -fache Dereferenzierung von *v* eine konstante Variable ist. Betrachten wir die folgenden Deklarationen:

```
1 int          e = 23;  
2 int  const   f = 42;  
3 int      *const pe = &e;  
4 int  const *const pf = &f;  
5 int      *const *ppe = &pe;  
6 int  const *const *ppf = &pf;
```



Hier werden deklariert:

1. Eine nicht-konstante Variable **e**.
2. Eine konstante Variable **f**.
3. Ein konstanter Pointer **pe** auf eine nicht-konstante Variable.
4. Ein konstanter Pointer **pf** auf eine konstante Variable.
5. Ein nicht-konstanter Pointer **ppe** auf einen konstanten Pointer auf eine nicht-konstante Variable.
6. Ein nicht-konstanter Pointer **ppf** auf einen konstanten Pointer auf eine konstante Variable.

Ohne Warnung kompilieren also die folgenden Statements:

```
1 ppf++;  
2 (**ppe)++;  
3 e++;  
4 (*pe)++;
```

Während die folgenden eine Warnung erzeugen:

```
1 pf++;  
2 (**ppf)++;  
3 f++;
```

## 8.3 Funktionenpointer

Funktionen werden vom Compiler in Maschinencode übersetzt, der letzten Endes bei der Ausführung auch im Speicher liegt, und dort von der CPU als Befehle interpretiert wird. Daher haben auch Funktionen bei der Ausführung eine Adresse im Speicher. In C ist es möglich, die Adresse einer Funktion zu ermitteln und diese in einer Variablen zu speichern – in einem sogenannten *Funktionenpointer*. Einen Funktionenpointer deklariert man wie folgt:

**RÜCKGABETYP** (**\*NAME**) (**PARAMETERTYP1**, ..., **PARAMETERTYPn**);

Als Beispiel würde die variable **function** im folgenden Programmcode eine Variable sein, die Pointer auf Funktionen speichert, welche zwei **double**-Variablen als Argumente erwarten und ein **double** als Rückgabewert liefern. Der Hit: Wir können diesen Pointer danach wieder dereferenzieren und die referenzierte Funktion aufrufen!

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     /* Deklaration des Funktionenpointers: */
6     double (*function) (double,double);
7     /* Wir weisen function die Adresse von pow zu: */
8     function = &pow;
9     /* Nun dereferenzieren wir function, erhalten
10        damit pow zurück und rufen es auf: */
11    printf("%f\n", (*function)(2.0,3.0));
12    return 0;
13 }

```

Damit nicht genug, wir wollen Funktionenpointer an andere Funktionen übergeben. Im folgenden Beispiel beschreiben wir eine Funktion, welche die Trapezsumme einer Funktion  $f : \mathbb{R} \rightarrow \mathbb{R}$  im Intervall  $[a, b]$  berechnet:

```

1 #include <stdio.h>
2 #include <math.h>
3
4 /* Der Einfachheit halber definieren wir einen neuen
5     Typ fuer Funktionen, die ein double erwarten und
6     ein double liefern: */
7 typedef double (*REALFUNCTION) (double);
8
9 double Trapez(REALFUNCTION f, double a, double b) {
10     if (a > b) return Trapez(f,b,a);
11     return (b-a) * ( (*f)(b) + (*f)(a) ) / 2;
12 }
13
14 int main() {
15     printf("%f\n", Trapez(&exp,1,3) );
16     printf("%f\n", Trapez(&log,1,3) );
17     printf("%f\n", Trapez(&sin,1,3) );
18     return 0;
19 }

```

Mit Hilfe von Funktionenpointern können wir die `qsort`-Funktion aus `<stdlib.h>` verwenden:

```

1 void qsort(
2     void          *array,
3     unsigned long count,
4     unsigned long size,
5     int (*compare) (const void *a, const void *b)
6 );

```

Diese Funktion erwartet an der Adresse `array` genau `count` Elemente, die jeweils

`size` viele Speicherzellen in Anspruch nehmen. Der Funktionspointer `compare` verweist auf eine Funktion, mit der zwei solche Elemente verglichen werden können. Dabei wird die Rückgabe von `compare` wie folgt interpretiert:

Rückgabe	Bedeutung
0	Die Elemente gelten als "gleich".
1	Das Element bei <code>a</code> ist "größer" als das bei <code>b</code> .
-1	Das Element bei <code>a</code> ist "kleiner" als das bei <code>b</code> .

Mit Hilfe dieser Informationen sortiert `qsort` das Array bei `array` aufsteigend. Ein einfaches Beispiel, um ein `int`-array absteigend zu sortieren:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int absteigen(const void *a, const void *b) {
5     int da = *((int const *)a);
6     int db = *((int const *)b);
7     if (da < db) return 1;
8     else if (db < da) return -1;
9     else return 0;
10 }
11
12 int main() {
13     int i;
14     int a[10] = { 23, 1, 23, 576, 3, 97, 7, 743, 2, 98 };
15     qsort( a, 10, sizeof(*a), &absteigen );
16     for (i=0; i<10; i++) printf("%i ", a[i]);
17     return 0;
18 }
```

## 9 Multithreading mit OpenMP

Diese Kapitel soll den Einstieg in Thread-Programmierung erleichtern, kann dabei natürlich weder die Vollständigkeit einer Dokumentation bieten noch die Erfahrung ersetzen, die man für Thread-Programmierung benötigt.

In diesem Kapitel möchten wir eine Bibliothek vorstellen mit der in C sogenanntes “Threads” realisiert sind. Threads ermöglichen es, mehrere Dinge gleichzeitig zu berechnen. In alten Heimcomputern ist dies nicht möglich – das höchste der Gefühle ist hier, zwei Berechnungen sehr schnell abwechselnd durchzuführen. In modernen Computern oder sogar in Computersystemen, die für paralleles Rechnen ausgelegt sind, stehen mehrere CPUs (oder CPUs mit mehreren Kernen) zur Verfügung. Alle C-Programme, die wir bisher geschrieben haben, nutzen maximal eine CPU bzw. maximal einen CPU-Kern. Die Idee beim Threading ist, neben dem Hauptprogramm (ab jetzt *Masterthread* genannt) weitere Nebenprogramme zu starten (ab jetzt *Workerthreads* genannt). Den Masterthread und die Workerthreads zusammen nennt man das *Threadteam*.

Um die technische Realisierung wird sich die Bibliothek “OpenMP” kümmern, wir wollen anhand des folgenden Beispiels zeigen, wie sie verwendet werden kann:

```
1 int main() {  
2     int i;  
3     double a[10000];  
4     for(i=0; i<10000; i++) {  
5         a[i] = i*i / 2.0;  
6     }  
7     return 0;  
8 }
```

In diesem einfachen Programm wird ein Array mit den halben Quadratzahlen gefüllt. Zu beachten ist, dass die Berechnung der nächsten Quadratzahl nicht von der vorhergehenden abhängt, also können wir durch das Hinzufügen von nur zwei Zeilen die Berechnung parallelisieren:

```
1 #include <omp.h>  
2 int main() {  
3     int i;  
4     double a[10000];  
5     #pragma omp parallel for  
6     for(i=0; i<10000; i++) {  
7         a[i] = i*i / 2.0;  
8     }  
9     return 0;  
10 }
```

Verwendet man gcc, muss beim Kompilieren das Compiler-Flag `-fopenmp` angegeben werden. Es weist den gcc dazu an, das “Compiler-Plugin” `openmp` zu verwenden. Führt man das Programm danach aus, erzeugt der Masterthread in Zeile 5 einige

Workerthreads und die Berechnung in der `for`-Schleife wird auf sie aufgeteilt – diesen Vorgang nennt man *Fork*. In Zeile 8 ist die Schleife zuende und die Workerthreads verschwinden wieder – dies nennt man *Join*.

Die Anweisung `#pragma omp parallel for` ist eine Kurzschreibweise für eine andere Anweisung, diese werden wir zunächst kennen lernen. Solche Anweisungen von OpenMP haben die folgende Form:

```
#pragma omp DIRECTIVE CLAUSES
```

Das weitere Ziel ist nun, Worte zu lernen, die als DIRECTIVE und CLAUSES angegeben werden können (mehrer Klauseln werden durch Leerzeichen oder Kommata getrennt).

## 9.1 Forking und Joining

Um an irgendeiner Stelle zu Forken – also den Masterthread anzuweisen Workerthreads zu erzeugen – verwendet man die Direktive `parallel`. Danach kann ein C-Block angegeben werden, der dann von allen Threads gleichermaßen ausgeführt wird:

```
1 #pragma omp parallel
2 {
3     do_work();
4 }
```

In diesem Beispiel wird die Funktion `do_work` von allen Threads gleichermaßen aufgerufen. Innerhalb des `parallel`-Block können nun verschiedene weitere Direktiven angegeben werden um dafür zu sorgen, dass nicht alle Threads exakt das gleiche. Ein naiver Ansatz wäre etwa die Verwendung der *Laufzeitfunktion* `omp_get_thread_num`, die in `<omp.h>` deklariert ist und die eindeutige Nummer des gerade aktiven Threads zurück gibt.

```
1 #pragma omp parallel thread_num(2)
2 {
3     if (omp_get_thread_num() == 0) do_work();
4     else do_other_work();
5 }
```

Listing 42: Verwendung der Parallel-Direktive

In diesem Beispiel wurde ausserdem die Klausel `thread_num(ANZAHL)` verwendet mit der man die Anzahl der Threads steuern kann. Alternativ kann man auch die Umgebungsvariable `OMP_NUM_THREADS` setzen um die Anzahl der Threads fest zu legen. Wird beides nicht gemacht, entscheidet OpenMP über die beste Anzahl von Threads (liegt nur eine CPU mit nur einem Kern vor, beträgt diese wahrscheinlich 1).

Eine weitere Klausel ist `if`, sie nimmt eine Expression als Argument entgegen. Wenn die Expression zu wahr auswertet, wird der Fork tatsächlich ausgeführt, sonst wird einfach mit einem einzigen Thread, dem Masterthread fortgefahren. Dies ist z.B. dann nützlich, wenn sich herausgestellt hat, dass sich Parallelisierung erst ab einer bestimmten Datengröße lohnt und das Programm vorher, wegen des Verwaltungsaufwand, der durch Threads verursacht wird, sogar langsamer wird.

## 9.2 Sektionen

Möchte man, wie im Listing 42 aus dem letzten Kapitel, zwei Aufgaben an zwei Threads verteilen, kann die Direktive `sections` verwendet werden. Im darauffolgenden Block, kann mehrmals die Direktive `section` angegeben werden. Erreicht das Programm den `sections` Block, werden die eingeschlossenen `sections` unter den bestehenden Threads aufgeteilt. Gibt es mehr `sections` als Threads, wird ein Thread mehrere `sections` nacheinander ausführen, gibt es mehr Threads als `sections`, haben einige Threads nichts zu tun.

```
1 #pragma omp sections
2 {
3   #pragma omp section
4   {
5       do_work();
6   }
7   #pragma omp section
8   {
9       do_other_work();
10  }
11 }
```

Die Direktive `sections` sollte natürlich nur innerhalb eines `parallel`-Blocks angegeben werden, sonst werden die `sections` nur unter dem Masterthread “aufgeteilt”, der dann einfach die gesamte Arbeit macht.

## 9.3 Schleifen

Zum parallellisieren einer For-Schleife, verwendet man die Direktive `for`

```
1 #pragma omp for
2 for(i=0; i<10000; i++) {
3     a[i] = i*i / 2.0;
4 }
```

Wir möchten an dieser Stelle erneut darauf hinweisen, dass eine wichtige Eigenschaft des Codes innerhalb der For-Schleife darin besteht, dass kein Durchlauf von einem vorhergehenden abhängt: Erreicht das Threadteam nämlich die `for`-Direktive, steht nicht fest, mit welchem Schleifendurchlauf begonnen wird. Es

könnte sehr gut sein, dass der Durchlauf für `i=1` *vor* dem Durchlauf von `i=0` ausgeführt wird. Folgender Code würde ohne Parallelisierung korrekt funktionieren, *mit* aber nicht:

```
1 a[0]=1
2 #pragma omp for
3 for(i=1; i<n; i++) {
4     a[i] = a[i-1] * 2.0; /* DO NOT TRY THIS AT HOME */
5 }
```

Ein großes Problem an solchem Code ist ausserdem, dass er sehr wohl wie erwartet funktionieren *kann* (z.B. wenn man ihn Zuhause testet, nur eine CPU mit nur einem Kern hat und die Sterne richtig stehen). Es kann aber ganz plötzlich dazu kommen, dass er unerwartete Resultate liefert, weil beispielsweise der Durchlauf von `i=2` vor dem Durchlauf von `i=1` ausgeführt wird und `a[1]` darum noch undefiniert ist. Die oben stehende Schleife lässt sich also einfach *nicht parallelisieren*.

Es sind auch nicht alle Formen von For-Schleifen erlaubt, beispielsweise muss zu Beginn der Schleife bereits feststehen, wie oft sie ausgeführt wird, damit es überhaupt möglich ist, die verschiedenen Schleifendurchläufe auf die Threads aufzuteilen.

## 9.4 Barrieren

Erreicht ein Thread des Teams eine mit der **barrier**-Direktive definierte Barriere, wartet er darauf, bis alle anderen Threads auch diese Stelle erreicht haben:

```
1 #pragma omp parallel
2 {
3     do_work1();
4 #pragma omp barrier
5     do_work_that_depends_on_work1();
6 }
```

Listing 43: Barriere

Sowohl hinter der **sections**-Direktive als auch hinter der **for**-Direktive befindet sich eine sogenannte *implizite Barriere*, das bedeutet, dass nach Beenden des entsprechenden Blockes mit der weiteren Ausführung auf das gesamte Threadteam gewartet wird. Mit der Klausel **nowait** kann dieses Verhalten geändert und die implizite Barriere entfernt werden:

```
1 #pragma omp parallel
2     f = 2.0;
3 #pragma omp for nowait
4     for(i=0; i<n; i++) {
5         z[i] = x[i] + y[i] * f;
```

```

6|     }
7| #pragma omp for nowait
8|     for(i=0; i<n; i++) {
9|         a[i] = b[i] + c[i];
10|    }
11| #pragma omp barrier
12|    sum = 0;
13|    for(i=0; i<n; i++) {
14|        sum += z[i] * a[i];
15|    }
16|    do_something_with(sum);
17| }

```

Listing 44: Explizierte Barriere

In diesem Beispiel wurde in Zeile 11 eine explizierte Barriere eingefügt, da sonst nicht sichergestellt ist, dass die Arrays `z` und `a` schon vollständig berechnet wurden. Hier sei darauf hingewiesen, dass die Berechnung von `sum` von jedem Thread einzeln ausgeführt wird. Es wäre nun naheliegend diese Berechnung auch zu parallelisieren, dabei ist allerdings Vorsicht geboten! Mehr dazu in [9.5](#).

## 9.5 Shared Memory

Bisher haben wir noch kein Wort dazu verloren, was beim Erreichen eines `parallel-`Blocks mit den bis dahin deklarierten Variablen geschieht. Es gibt zwei nahe-  
liegende Verhaltensweisen:

**private Variable** Man übergibt der Klausel `private` eine Komma-getrennte Liste von Variablen, die als “privat” ausgewiesen werden sollen. Jeder Thread erhält dann eine Variable von gleichem Namen und Typ. Ändert der Thread sie, sind diese Änderung nur für ihn selbst gültig. Standardmäßig sind `private` Variablen für jeden Thread uninitialisiert.

Möchte man sie mit dem Wert der Variablen vor dem `fork` initialisieren, muss man sie an die Klausel `firstprivate` übergeben (man kann sie dann auch aus der Liste von `private` weg lassen).

Bei der `for-` bzw. `sections-`Direktive gibt es eine klar definierte “letzte Aufgabe”, nämlich den sequentiell letzten Schleifendurchlauf bzw. die lexikographisch letzte `section`. Möchte man den Wert einer privaten Variable dieser letzten Aufgabe beim `Join` übernehmen, muss man sie in der Variablenliste, die man an `lastprivate` übergibt, aufführen (auch hier kann die Variable aus der Liste von `private` weg gelassen werden).

**geteilte Variable / shared Variable** alle Threads teilen sich die gleiche Variable. Auch `shared` nimmt eine Liste von Variablen entgegen, die danach von allen Threads des Teams geteilt werden. Das Schreiben in geteilte Variablen



verursacht sehr viele Probleme bei Programmierung der mit Threads und sollte so oft wie möglich vermieden werden. Details dazu siehe in ??.

Die `default`-Klausel der `parallel`-Direktive nimmt ein Argument entgegen, es ist entweder `none` oder `shared` (ist die Klausel nicht angegeben ist dies der Standard). Die Angabe von `default(shared)` bedeutet, dass Variablen, die weder explizit als `private` oder `geteilt` ausgewiesen werden als `shared` Variables betrachtet werden. Gibt man hingegen `default(none)` an, so müssen Variablen die innerhalb des `parallel`-Blocks verwendet werden explizit als “privat” oder “geteilt” ausgewiesen werden. Um versehentliches Schreiben in eine geteilte Variable zu vermeiden empfehlen wir darum immer die Angabe von `default(none)`.

Um nun die Berechnung der Summe aus Listing 44 zu parallelisieren, könnte man folgenden naiven Ansatz verfolgen:

```
1 sum = 0;
2 #pragma omp parallel shared(sum)
3     #pragma omp for
4     for(i=0; i<n; i++) {
5         sum += z[i] * a[i]; /* BAAAD! */
6     }
7 }
```

Listing 45: Falsche Art eine Summenberechnung zu parallelisieren

Das Problem tritt beim Schreiben in die geteilte Variable `sum` auf: Es kann sein, dass Thread A den Wert von `sum` aus dem Speicher liest, dann Thread B das gleiche macht. Jetzt erst addiert Thread A den Wert `z[i] * a[i]` zu `sum` und speichert ihn wieder im Speicher. Danach macht Thread B das gleiche und überschreibt den Wert, den Thread A gespeichert hat, wir haben also einen Summanden verloren. Solche Probleme löst man im Allgemeinen mit Locks (siehe 9.7) oder der `critical`-Direktive (siehe 9.6), in diesem speziellen Fall stellt OpenMP aber eine sehr elegante Methode zur Verfügung: die `reduction`-Klausel. Sie nimmt einen Operator und, mit einem Doppelpunkt davon getrennt, eine Liste von Variablen entgegen. Alle übergebenen Variablen werden als privat ausgewiesen und, je nach Operator, initialisiert.

Operator	Wert
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0

Tabelle 10: Erlaubte Operatoren und ihre Initialisierungswerte

Jeder Thread kann nun auf seine eigene Version der Variablen zugreifen und problemlos beschreiben. Nach Beendigung des entsprechenden Blockes werden dann die privaten Variablen aller Threads mit dem angegebenen Operator kombiniert. Eine notwendige Voraussetzung an den Operator muss es also sein, dass er kommutativ und assoziativ ist.

```

1 sum = 0;
2 #pragma omp parallel reduction(+:sum)
3   #pragma omp for
4     for(i=0; i<n; i++) {
5       sum += z[i] * a[i];
6     }
7 }
```

Listing 46: Parallelisierung einer Summenberechnung

## 9.6 single, master und critical

Ein hinter der **single**-Direktive angegebener Block wird nur von einem Thread ausgeführt. Wird die **nowait**-Klausel nicht angegeben, gibt es danach eine implizite Barriere.

```

1 do_work1(); /* wird von jedem Thread ausgeführt */
2 #pragma omp single
3 {
4     do_work2(); /* wird nur ein einziges Mal ausgeführt */
5 }
6 do_work3(); /* wird wieder von jedem Thread ausgeführt */
```

Listing 47: Ausführung von nur einem Thread

Die Direktive **master** verhält sich wie die **single**-Direktive mit folgenden Unterschieden:

- Der Thread, der die Aufgabe ausführen soll, ist immer der Master-Thread. Dies ermöglicht es beispielsweise auf den Wert einer privaten Variable über mehrere **master**-Blöcke hinweg zuzugreifen.
- Nach dem **master**-Block gibt es keine implizierte Barriere.

Schlussendlich ermöglicht es die **critical**-Direktive einen Block nur von exakt einem Thread *gleichzeitig* auszuführen. Im Gegensatz zu **single** und **master** wird er allerdings von jedem Thread ausgeführt. Die **critical**-Direktive erhält ein optionales Argument – ihren Namen. Gibt es an unterschiedlichen Stellen **critical**-Blöcke mit dem gleichen Namen, kann immer nur eine gleichzeitig ausgeführt werden. Erreicht ein Thread den Beginn eines **critical**-Blocks, während ein Block mit dem gleichen Namen bereits ausgeführt wird, wartet er.

```

1 #pragma omp critical(writeinfile)
2 {
3     write_data_to_file();
4 }
5 do_something();
6 #pragma omp critical(writeinfile)
7 {
8     write_data_to_file();
9 }

```

Listing 48: Ausführung von nur einem Thread

Das Argument für den Namen kann weggelassen werden, der Block erhält dann einen Standard-Namen. Es darf also auch nur ein **critical**-Block mit dem Standard-Namen ausgeführt werden.

Code-Bereiche, die immer nur von einem Thread ausgeführt werden dürfen (wie das Schreiben in eine Datei beispielsweise) nennt man *kritische Bereiche*, daher auch der Name der **critical**-Direktive. Manchmal ist die Definition von kritischen Bereichen über diese Direktive allerdings nicht flexibel genug: Immerhin darf über das gesamte Programm hinweg immer nur ein einziger **critical**-Block mit dem gleichen Namen ausgeführt werden. Es ist aber auch denkbar, dass zwei Threadteams, die die gleiche Funktion aufrufen sich gegenseitig nicht behindern sollen. In solchen Situationen verwendet man *Locks*.

## 9.7 Locks

Ein Lock ist eine Variable vom Typ `omp_lock_t`, welcher in `omp.h` definiert ist. Ein Lock hat zwei Zustände: entweder “offen” oder “geschlossen”, was dafür verwendet werden kann, sehr flexible kritische Sektionen zu definieren. In `omp.h` werden dazu noch folgende Funktionen definiert:

**omp\_init\_lock(\*omp\_lock\_t)** Initialisiert eine Lock-Variable, muss vor Benutzung eines Locks aufgerufen werden. Nach Initialisierung ist eine Lock-Variable offen.

**omp\_destroy\_lock(\*omp\_lock\_t)** Zerstört eine Lock-Variable wieder, diese Funktion muss mit jeder Lock-Variablen aufgerufen werden. Vor der Zerstörung muss die Variable offen sein.

**omp\_set\_lock(\*omp\_lock\_t)** versucht eine Lock-Variable zu schließen, ist sie bereits geschlossen, so wartet der aktuelle Thread mit der Ausführung (dieses Verhalten nennt man auch *blocking*) bis die Variable wieder offen ist und wiederholt den Vorgang.

**omp\_unset\_lock(\*omp\_lock\_t)** öffnet die Lock-Variable. Die Lock-Variable sollte vom gleichen Thread wieder geöffnet werden, der sie auch geschlossen hat.

**omp\_test\_lock(\*omp\_lock\_t)** versucht eine Lock-Variable zu schließen. Ist sie bereits geschlossen, gibt die Funktion 0 zurück. Ansonsten schließt sie das Lock und gibt 1 zurück. (Dieses Verhalten nennt man auch *non-blocking*).

Eine kritische Sektion lässt sich nun beispielsweise so realisieren:

```
1 omp_lock_t write_to_file_lock;
2 omp_init_lock(&write_to_file_lock);
3
4 #pragma omp parallel
5 {
6     do_something1();
7
8     omp_set_lock(&write_to_file_lock);
9     write_to_file();
10    omp_unset_lock(&write_to_file_lock);
11
12    do_something2();
13
14    while(!omp_test_lock(&write_to_file_lock)) {
15        do_something_else();
16    }
17    write_to_file();
18    omp_unset_lock(&write_to_file_lock);
19
20    do_something3();
21 }
22 omp_destroy_lock(&write_to_file_lock);
```

Listing 49: Kritische Sektion mit Locks

## A Referenzen

Es folgen Referenzen für einige wichtige Systemmodule, welche C bereits zur Verfügung stellt.

### A.1 Referenz `<math.h>`

Durch einbinden der Systemheaderdatei `<math.h>` stehen die folgenden mathematischen Funktionen zur Verfügung:

Deklaration	Rückgabewert
<code>double acos(double x);</code>	Arcuscosinus von <code>x</code>
<code>double asin(double x);</code>	Arcussinus von <code>x</code>
<code>double atan(double x);</code>	Arcustangenz von <code>x</code>
<code>double atan2(double y, double x);</code>	Arcustangenz von <code>x/y</code>
<code>double ceil(double x);</code>	Kleinste ganze Zahl, welche $\geq x$ ist
<code>double cos(double x);</code>	Cosinus von <code>x</code>
<code>double cosh(double x);</code>	Cosinus hyperbolicus von <code>x</code>
<code>double exp(double x);</code>	$e^x$ , wobei $e = 2.718281\dots$
<code>double fabs(double x);</code>	$ x $ (Betrag von <code>x</code> )
<code>double floor(double x);</code>	Größte ganze Zahl, welche $\leq x$ ist
<code>double ldexp(double x, double y);</code>	$x \cdot 2^y$
<code>double log(double x);</code>	Natürlicher Logarithmus von <code>x</code>
<code>double log10(double x);</code>	Zehnerlogarithmus von <code>x</code>
<code>double pow(double x, double y);</code>	$x^y$
<code>double sin(double x);</code>	Sinus von <code>x</code>
<code>double sinh(double x);</code>	Sinus hyperbolicus von <code>x</code>
<code>double sqrt(double x);</code>	$\sqrt{x}$ (Quadratwurzel aus <code>x</code> )
<code>double tan(double x);</code>	Tangens von <code>x</code>
<code>double tanh(double x);</code>	Tangens hyperbolicus von <code>x</code>

Tabelle 11: Von `math.h` exportierte Funktionen

Außerdem ist in `<math.h>` eine Konstante `HUGE_VAL` definiert: Dies ist der größte Wert, den eine Variable vom Typ `double` annehmen kann. Fast immer entspricht dies einem symbolischen Wert “ $\infty$ ”.

## A.2 Referenz <time.h>

Zunächst sind in der <time.h> einige neue Datentypen definiert, welche in der Regel nur Umbenennungen (siehe 7.4) von gewöhnlichen, ganzzahligen Datentypen sind:

Datentyp	Bedeutung
clock_t	Speichert <i>CPU-Zeiten</i> (siehe unten)
size_t	Speichert ganzzahlige Größenangaben
time_t	Datentyp für Zeitangaben (meistens eine Anzahl von Sekunden)

Tabelle 12: Einfache Datentypen aus time.h

Eine CPU braucht für jeden Maschinenbefehl die gleiche Zeit. Dieses Zeitintervall wird auch als *Takt* bezeichnet, und eine CPU-Zeit entspricht einer gewissen Anzahl Takte.

Weiterhin definiert ist die Struktur **struct tm**, welche alle Komponenten einer Kalenderzeit im Gregorianischen Kalender enthält. Ihre Definition muss mindestens die folgenden Felder enthalten:

```

1 struct tm {           /* Felddescription           Intervall */
2                       /* ----- */
3   int tm_sec;         /* Sekunden nach der Minute   [0,59] */
4   int tm_min;         /* Minuten nach der Stunde    [0,59] */
5   int tm_hour;        /* Stunden seit Mitternacht   [0,23] */
6   int tm_mday;        /* Tag des Monats             [1,31] */
7   int tm_mon;         /* Monat seit Januar          [0,11] */
8   int tm_year;        /* Jahreszahl                 [1900,[ */
9   int tm_wday;        /* Tag seit Sonntag           [0,6] */
10  int tm_yday;        /* Tag seit dem 1. Januar     [0,365] */
11                      /* ----- */
12  int tm_isdst;       /* Zeigt an, ob es sich um Sommerzeit */
13                      /* (daylight saving time) handelt.    */
14                      /* 0 steht hierbei für "Nein", -1 für */
15                      /* "unbekannt", alles andere für "Ja" */
16 };

```

Tabelle 13 gibt Aufschluss über die Funktionen zum Berechnen der Zeit, welche in <time.h> definiert sind:

Deklaration	Effekt
<code>time_t time(time_t *p);</code>	Gibt bei Misserfolg -1 zurück, andernfalls die momentane Zeit. Ist <code>p</code> $\neq$ NULL, so wird dieses Ergebnis in <code>*p</code> zusätzlich noch abgespeichert.
<code>clock_t clock();</code>	Gibt bei Misserfolg -1 zurück, andernfalls die seit Programmstart vergangene CPU-Zeit.

Tabelle 13: Funktionen zum Ermitteln der Zeit

Die folgenden Umrechnungsfunktionen geben stets den gleichen Pointer auf einen internen `struct tm` zurück, welcher *auf keinen Fall* vom Programmierer freigegeben werden sollte. Des weiteren überschreibt jeder Aufruf einer Umrechnungsfunktion die Werte dieser Struktur:

Deklaration	Umrechnung in
<code>struct tm *gmtime(time_t*);</code>	UTC-Kalenderzeit
<code>struct tm *localtime(time_t*);</code>	Lokale Kalenderzeit

Tabelle 14: Umrechnungsfunktionen für Zeiten

Um die Funktion `clock()` sinnvoll zu verwenden, ist in der `<time.h>` noch die Konstante `CLOCKS_PER_SEC` definiert, welche die Anzahl CPU-Takte pro Sekunde angibt.

### A.3 Referenz <stdlib.h>

Zunächst ist in der <stdlib.h> der Datentyp `size_t` (siehe dazu auch Tabelle 12 in Anhang A.2) und die Konstante `NULL` definiert. Weiterhin stehen durch einbinden von <stdlib.h> die folgenden Funktionen zur Verfügung:

Deklaration	Funktionsweise
<code>void *malloc(size_t c);</code>	Siehe 5.2.
<code>void *realloc(void *p, size_t c);</code>	Siehe 5.5.
<code>void free(void *p);</code>	Siehe 5.4.
<code>int system(char *s);</code>	Hat den gleichen Effekt, als wäre die Zeichenfolge <code>s</code> auf der Kommandozeile eingegeben worden.
<code>int abs(int i);</code>	Berechnet den Absolutwert von <code>i</code> .
<code>long labs(long i);</code>	Berechnet den Absolutwert von <code>i</code> .
<code>int atoi(char *s);</code>	Gibt die Ganzzahl zurück, die in Dezimalschreibweise in <code>s</code> steht.
<code>double atof(char *s);</code>	Gibt die Gleitkommazahl zurück, die in <code>s</code> steht.

Tabelle 15: Funktionen aus `stdlib.h`

Des Weiteren stehen zur Verfügung:

Deklaration	Effekt
<code>void srand(unsigned s);</code>	Initialisiert den internen Zufallsgenerator mit dem Wert <code>s</code> .
<code>int rand();</code>	Liefert eine Pseudo-Zufallszahl aus dem Bereich von 0 bis <code>RAND_MAX</code> (wobei <code>RAND_MAX</code> eine Konstante ist, die garantiert $\geq 32767$ ist).

Tabelle 16: Funktionen für Zufallszahlen aus `stdlib.h`

Der Zufallszahlengenerator muss per `srand` initialisiert werden. Wird `rand` ohne vorherige Initialisierung aufgerufen, sind die zurückgegebenen Zufallszahlen bei jedem Programmablauf die gleichen (und damit nicht wirklich zufällig). Für gewöhnlich verwendet man die Funktion `time` (siehe Anhang A.2), um den Zufallszahlengenerator mittels

```
1 srand( time( NULL ) );
```

zu initialisieren.



## A.4 Referenz <limits.h>

In der <limits.h> sind die folgenden Konstanten definiert, welche maximale und minimale Größe ganzzahliger Datentypen enthalten.

Konstante	Mindestwert	Bedeutung
CHAR_BIT	8	Anzahl Bits in einem Byte
SHRT_MIN	-32767	Minimalwert für <code>signed short</code>
SHRT_MAX	+32767	Maximalwert für <code>signed short</code>
USHRT_MAX	65535	Maximalwert für <code>unsigned short</code>
INT_MIN	-32767	Minimalwert für <code>signed int</code>
INT_MAX	+32767	Maximalwert für <code>signed int</code>
UINT_MAX	65535	Maximalwert für <code>unsigned int</code>
LONG_MIN	-2147483647	Minimalwert für <code>long int</code>
LONG_MAX	+2147483647	Maximalwert für <code>long int</code>
ULONG_MAX	4294967295	Maximalwert für <code>unsigned long int</code>

Tabelle 17: Limits für Ganzzahlen

In der Realität sind die Werte häufig betragsmäßig größer als der Mindestwert, insbesondere für `int` und `long int`.

## A.5 Referenz <float.h>

In der <float.h> sind die folgenden Konstanten definiert, welche im Zusammenhang mit Fließkommazahlen häufig von Bedeutung sind:

Konstante	Bedeutung
FLT_MAX	Größter, endlicher Wert für ein <code>float</code>
FLT_MIN	Kleinsten positiver, endlicher Wert für ein <code>float</code>
FLT_EPSILON	Kleinsten positiver <code>float</code> -Wert $\varepsilon$ , für den $1.0 + \varepsilon \neq 1.0$ gilt
DBL_MAX	Größter, endlicher Wert für ein <code>double</code>
DBL_MIN	Kleinsten positiver, endlicher Wert für ein <code>double</code>
DBL_EPSILON	Kleinsten positiver <code>double</code> -Wert $\varepsilon$ , für den $1.0 + \varepsilon \neq 1.0$ gilt

Tabelle 18: Limits und Konstanten für Fließkommazahlen

Die Konstante `DBL_EPSILON` eignet sich hervorragend als Fehlertoleranz bei numerischen Algorithmen.

## B Operatorpräzedenzen

Wenn beim Auswerten einer Expression in C nicht klar ist, wie geklammert werden muss, hält sich der Compiler an sogenannte Operatorpräzedenzen. Operatoren mit höherer Präzedenz werden vor denen mit niedrigerer Präzedenz angewandt: So haben etwa arithmetische Punkt-Operationen eine höhere Präzedenz als arithmetische Strich-Operationen (Punkt- vor Strichrechnung).

Im Folgenden sind die C-Operatoren ihrer Präzedenz nach absteigend sortiert. Operatoren gleicher Präzedenz sind durch vertikale Trennlinien gruppiert.

Operator	Wirkung	Erklärung
()	Funktionsaufruf	Seite 27
[]	Indizierung	Seite 41
.	Feldauswahl	Seite 63
->	Feldauswahl bei Pointern auf Strukturen	Seite 66
++	Postfix Inkrementierung	Seite 22
--	Postfix Dekrementierung	Seite 22
++	Präfix Inkrementierung	Seite 22
--	Präfix Dekrementierung	Seite 22
-	Unäres Minus	Der Wert von <code>-a</code> ist das Negative von <code>a</code> .
~	Bitweises Komplement	Der Wert von <code>~a</code> ist der Wert, den man erhält, wenn man alle Bits von <code>a</code> invertiert.
!	Logische Verneinung	Tabelle 2.7 auf Seite 24
( <u>TYP</u> )	Typecasting	Seite 21
*	Dereferenzierungsoperator	Seite 39
&	Adressoperator	Seite 39
sizeof	Größenbestimmung	Seite 41

*	Multiplikation	Tabelle 2.4 auf Seite 20
/	Division	
%	Modulo-Operation	
+	Addition	
-	Subtraktion	
>>	Bitweiser Rechtsshift	Bei $a \gg n$ sind die Bits von $a$ um $n$ Stellen nach rechts verschoben und von links mit Nullbits aufgefüllt. Es entspricht einer Ganzzahldivision durch $2^n$ .
<<	Bitweiser Linksshift	Ähnlich wie oben, $a \ll n$ entspricht einer Multiplikation von $a$ mit $2^n$ .
< <=	Vergleiche	Tabelle 2.7 auf Seite 24
> >=	(Größer/Kleiner)	
==	Vergleiche	Tabelle 2.7 auf Seite 24
!=	(Gleich/Ungleich)	
&	Bitweises Und	Verknüpft die Bits der beiden Operanden einzeln mit der jeweiligen Operation.
^	Bitweises, exklusives Oder	
	Bitweises, inklusives Oder	
&&	Logisches Und	Tabelle 2.7 auf Seite 24
	Logisches Oder	
?:	Bedingte Auswertung	Siehe Seite 72
=	Zuweisung	Seite 20
×=	Kurzschreibweisen	Seite 21
,	Komma	Seite 22

#### Anmerkungen:

Die Operanden aller binären Operationen werden von links nach rechts ausgewertet, mit Ausnahme des *Zuweisungsoperators*: Hier wird von rechts nach links ausgewertet.

Eine Postfix-Inkrementierung bzw. -Dekrementierung darf vom Compiler bis zum nächsten *Statement* verzögert werden und muss nicht durchgeführt werden, sobald die entsprechende Expression ausgewertet wird. Die Rechnung  $y = x * x++$  würde also unter Umständen nicht  $(x + 1) \cdot x$ , sondern  $x^2$  berechnen und in  $y$  speichern.