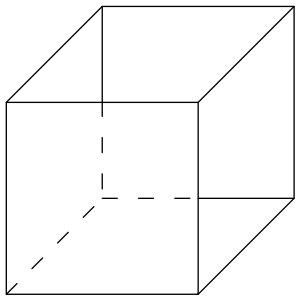
Skript zum Kompaktkurs
C mit Beispielen in OpenGL

Martin Kraus
Manfred Weiler
Dirc Rose
Friedemann Rößler

Institut für Visualisierung
und Interaktive Systeme,
Institut für Informatik,
Universität Stuttgart

Zweite überarbeitete Auflage
Sommersemester 2006
(basierend auf der Orginalfassung
vom Sommersemester 2002)



Für Tom

Inhaltsverzeichnis

Vorbemerkungen zur ersten Auflage

Dies ist ein Skript zu einem C-Kompaktkurs, der im Sommersemster 2002 an der Universität Stuttgart gehalten wurde. Es ist *nicht* der Kurs selbst. Der Kurs ist viel besser, weil auch Fragen gestellt werden können! Dieses Skript ist also nur eine Ergänzung, vor allem zum Nachschlagen. Trotzdem ist es parallel zum Kurs aufgebaut, damit die Orientierung leichter fällt.

Ich danke allen Beteiligten, insbesondere Manfred Weiler für Kapitel 8 und Dirc Rose für Kapitel 9 dieses Skripts. Besonderer Dank auch an Simon Stegmaier für's Korrekturlesen und die Betreuung der Übungen.

Martin Kraus, Stuttgart im Juli 2002

Vorbemerkungen zur zweiten Auflage

Der Aufbau und Inhalt des C-Kompaktkurses wurde zum Sommersemester 2006 teilweise überarbeitet. Inhaltlich wurden nur wenige Ergänzungen vorgenommen. Größere Umstellungen gab es dahingegen bei der Reihenfolge in der die einzelenen Themen behandelt werden und bei den Übungsaufgaben.

Ich möchte all meinen Vorgängern danken, die diesen Kurs konzipiert haben und ein sehr gutes Skript dazu verfasst haben, insbesonder natürlich Martin Kraus. Mein besonderer Dank gilt auch Joachim Vollrath, der mir sehr viel Arbeit beim Erstellen der vorlesungsbegleitenden Folien abgnommen hat.

Friedemann Rößler, Stuttgart im Juli 2006

Teil I

Erster Tag

Kapitel 1

Einführung in C

1.1 Was ist ein C-Programm?

Ein „C-Programm“ ist ein in C geschriebenes Computerprogramm. Die meisten von Menschen geschriebenen C-Programme fangen mit einer einfachen ASCII-Textdatei an, deren Name auf `.c` endet und damit deutlich macht, dass sie in der Sprache C geschrieben wurde. Diese `.c`-Datei kann mit irgendeinem Texteditor wie emacs, xemacs, nedit, gedit, etc. geschrieben werden und wird mit Hilfe eines C-Compilers in ein ausführbares Programm übersetzt. D.h. die Befehle in der Programmiersprache C werden in eine Maschinensprache übersetzt, die der Prozessor versteht. Da jede Prozessorfamilie ihre eigene Maschinensprache spricht, muss es auch jeweils einen eigenen C-Compiler geben, und ein C-Programm muss für jeden Prozessor, auf dem es ausgeführt werden soll, neu übersetzt werden. Das ist sehr lästig und wird noch lästiger, wenn sich die C-Compiler leicht unterscheiden. Deswegen ist die Standardisierung von C eine so wichtige Sache.

C wurde in den 70er Jahren von Denis Ritchie und Brian W. Kernighan für UNIX entworfen. Diese ursprüngliche Form wird heute oft K&R-C genannt, aber kaum noch verwendet. C in seiner heutigen Form wird oft ANSI-C genannt und wurde 1989 definiert. Die ISO-Standardisierungen von 1990 und 1995 haben vergleichsweise wenig Änderungen mit sich gebracht. Der jüngste ISO-Standard wurde 1999 verabschiedet. Er bringt durchaus einige Änderungen mit sich, die für eine Einführung in C aber kaum relevant sind.

1.1.1 Das erste C-Programm

Bevor wir uns näher mit den einzelnen Elementen der Programmiersprache C auseinandersetzen, wollen wir zuächst an einem Beispiel, das den Text `Hallo World` ausgibt, den grundsätzlichen Aufbau eines C-Programms betrachten:

```
/* Standardbibliotheken einfuegen */
#include <stdio.h>

/* Beginn des Hauptprogramms */
int main(void)
{
   /* Aufruf der Standardfunktion
      printf zur Ausgabe von
      "Hallo World" */
   printf("Hallo World!");

   /* Erfolgreiche Ausfuehrung
      rueckmelden */
   return(0);
}
```

Jedes C-Programm besteht aus Funktionen. Notwendig für jedes C-Programm ist die Funktion `main()` Die `main()`-Funktion bildet das Hauptprogramm, alle anderen Funktionen sind mit Unterprogrammen vergleichbar, wie sie auch in anderen Programmiersprachen üblich sind.

Funktionen werden durch geschweifte Klammern in Blöcke eingeteilt. Zu jeder öffnenden Klammer {, die einen Block einleitet, muss eine schließende Klammer } gehören, die den Block beendet. Die Funktion `main` im obigen Beispiel besteht nur aus einem Block. Dieser äußere Block einer Funktion wird auch als Funktionsrumpf bezeichnet.

Innerhalb eines Block stehen die Anweisungen des C-Programms. Anweisungen werden in C durch Semikolon ; abgeschlossen. Ein einzelne Anweisung kann somit über mehrere Zeilen gehen.

Mit der Anweisung `printf(...)` erfolgt die Ausgabe einer Zeichenkette (hier `Hallo World!`) auf

dem Bildschirm. Hierbei handelt es sich genau genommen um den Funktionsaufruf einer Standardfunktion. Standardfunktionen sind fertige Funktionen, die zur C-Implementierung gehören.

Mit Hilfe der `return`-Anweisung wird dem Aufrufer zurückgemeldet, ob das Programm erfolgreich ausgeführt wurde. Der Wert 0 gibt in der Regel die erfolgreiche Ausführung an. Werte ungleich 0 werden als Fehler interpretiert. Durch Rückgabe unterschiedlicher Fehlercodes, kann dem Aufrufer die Art des aufgetretenen Fehlers mitgeteilt werden.

Jede in einem Programm verwendete Funktion, muss deklariert werden. Dies gilt in der Regel auch für Standardfunktionen. Die Deklarationen der Standarfunktionen sind in sogenannten Header-Dateien zusammengefasst. Die Deklarationen für die Standardfunktionen der Ein- und Ausgabe, unter anderen `printf`, sind beispielsweise in der Headerfunktion `stdio.h` enthalten. Mit der ersten Zeile des obigen Beispielprogramms `#include <stdio.h>` wird die Header-Datei `stdio.h` eingefügt und so die darin deklarierten Funktionen dem Compiler bekannt gemacht.

Mit `/*` und `*/` werden Kommentare eingeschlossen. Diese werden vom C-Compiler ignoriert und sind deshalb nicht Teil des ausführbaren Programm. Trotzdem sind sie sehr wichtig, weil sie an beliebiger Stelle in einer `.c`-Datei Erklärungen, Hinweise, etc. geben können und so fest mit dem C-Code verbunden sind, dass sie normalerweise nur zusammen mit dem C-Code verloren gehen. Wie im Beispielprogramm zu sehen ist, können Kommentare auch über mehrere Zeilen gehen. In C gilt fast immer, dass das Zeilenende keine Bedeutung hat, deswegen werden im Folgenden nur die Ausnahmen von dieser Regel erwähnt. Wichtig ist außerdem, dass C-Kommentare nicht geschachtelt werden können.

Die Beispiele in diesem Kurs sind so klein und im Text so detailliert beschrieben, dass meistens auf Kommentare verzichtet wird. Das sollte aber nicht der Normalfall sein!

Die im Beispielprogramm gewählte Art der Formatierung ist zwar in C üblich (und sinnvoll!), aber nicht notwendig. Grundsätzlich können C-Programme formatfrei geschrieben werden. Das obige Beispiel könnte daher auch folgendermaßen aussehen:

```
#include <stdio.h>
int main(void){
printf("Hallo World\n");}
```

Aber Einrückungen, Zeilenumbrüche usw/ erhöhen - genauso wie Kommentare - die Übersichtlickeit und Lesbarkeit von Programmen und es ist guter Programmierstil Programme mit ausreichend Kommentaren zu versehen und durch Formatierungen die Struktur des Programmablaufs deutlich zu machen

1.1.2 Übersetzen und Ausführen des Programms

Um unser erstes C-Programm ausführen zu können müssen wir es, wie oben bereits erläutert, zunächst ind Maschinencode übersetzen. Eigentlich muss man hierfür nur den C-Compiler aufrufen. Unter UNIX-Varianten klappt das meistens mit `cc` (Default-C-Compiler) bzw. `gcc` (GNU-C-Compiler). Dem Compiler muss man noch die `.c`-Datei angeben. Wurde das Programm beispielsweise in eine Textdatei `halloWorld.c` geschrieben, dann compiliert der Compileraufruf `gcc halloWorld.c` das Programm. Nach dem erfolgreichen Compilieren ist das Ergebnis ein Programm namens `a.out` im gleichen Verzeichnis und kann mit `./a.out` gestartet werden kann.

Möchte man das fertige Programm anders als a.out nennen, kann man mit der Kommandozeilenoption -o einen anderen Namen angeben. Zum Beispiel steht mit dem Befehl `gcc halloWorld.c -o halloWorld`, das ausführbare Programm nachher in der Datei `halloWorld`.

C-Compiler haben normalerweise viele weitere Kommandozeilenoptionen, z. B. `-ansi` um sicherzustellen, dass nur Programme nach ANSI-C klaglos compiliert werden, oder `-Wall` um alle Warnungen des Compilers auszugeben (was *sehr* sinnvoll ist). Allerdings sind diese Kommandozeilenoptionen meistens abhängig vom Compiler; mehr Informationen sind z. B. mit `man gcc` oder `man cc` zu erhalten.

1.2 Definitionen von Variablen und Zuweisungen

Eine Variable ist ein Stück Speicherplatz des Computers, der zum Speichern eines Wertes verwendet wird,

und dem ein Namen gegeben wurde (der Variablenname), um sich auf den aktuellen Wert der Variable (also das, was gerade im Speicher steht) beziehen zu können. Daraus folgen schon drei wesentliche Eigenschaften von Variablen:

- Sie haben eine Adresse, die sagt, wo sie im Speicher zu finden sind.
- Sie haben einen Typ, der sagt, wie die Bits (im Speicher eines Computers gibt es nur Bits) an der Speicherplatzadresse zu interpretieren sind. (Text, ganze Zahl, Gleitkommazahl, etc.)
- Jede Variable hat außerdem eine bestimmte Anzahl von Bytes, die der Variablenwert im Speicher belegt. In C wird diese Größe immer durch den Typ festgelegt, d.h. alle Variablen eines Typs brauchen gleichviele Bytes. Deswegen spricht man auch von der Bytegröße des Typs.

Variablen sind ein sehr grundlegendes Konzept. Rechenmaschinen ohne Variablen sind ziemlich undenkbar. Variablen erlauben es insbesondere Programme zu schreiben, die viele Fälle (viele verschiedene Werte der Variablen) behandeln können, ohne für jeden Fall ein eigenes Programm schreiben zu müssen. Daraus folgt auch: Sobald in einem Programm mehrmals sehr ähnliche Programmteile auftauchen, sollte man überlegen, ob man die Programmteile nicht mit Hilfe von Variablen zu einem zusammenfassen kann, der alle Fälle behandelt.

In C müssen Variablen immer erst einmal „definiert“ werden, bevor sie verwendet werden. Dabei wird

- der Typ der Variable und
- der Name der Variablen festgelegt und
- zur Laufzeit des Programms ein Stück Speicherplatz für den Wert der Variable reserviert (wieviele Bytes Speicherplatz benötigt werden, ergibt sich aus dem Typ).

Die einfachsten (und häufigsten) Variablendefinitionen folgen diesem Schema:

Datentyp Variablenname`;`

Zum Beispiel definiert

```
int i;
```

eine Variable mit Namen `i` vom Typ `int`. D.h. es wird ein Stück Speicherplatz reserviert, dessen Inhalt im folgenden Programmteil unter dem Namen `i` benutzt werden kann.

In der Praxis ist der häufigste Fehler bei der Variablendefinition ein vergessener Strichpunkt. Wenn er fehlt, verwirrt das den Compiler gelegentlich so sehr, dass er eine unsinnige Fehlermeldung ausgibt, in der normalerweise auch noch die falsche Zeilennummer angegeben wird. Bei Syntax-Fehlermeldungen des Compilers gilt allgemein, dass man zunächst mal den Text der Meldung ignorieren und in den Zeilen *vor* der vom Compiler bemängelten Zeile nach einem Fehler suchen sollte. (Im Übrigen ist die Interpretation von Compiler-Fehlermeldungen sowieso eine Kunst für sich.)

Vor dem Datentyp können noch Modifizierer stehen: `extern` und `static` werden später erklärt; `const` definiert eine unveränderliche Variable (wird in C selten genutzt, Konstanten werden häufiger mit Hilfe des C-Präprozessors Namen gegeben, was auch später beschrieben wird); `auto`, `register`, `restrict` und `volatile` sind auch kaum von Bedeutung.

1.2.1 Variablennamen

... werden auch *Identifier* genannt. Das erste Zeichen muss ein Buchstabe (`a`, ..., `z`, `A`, ..., `Z`) oder Unterstrich (`_`) sein, darauffolgende Zeichen dürfen auch Ziffern (`0`, ..., `9`) sein. Groß- und Kleinschreibung wird unterschieden. Beispiele: `i`, `index`, `Index`, `index5`, `ist_leer`, `laenge_des_vektors`, `laengeDesVektors`. Anmerkung: In diesem Skript werden deutsche Variablennamen, Kommentare und Funktionsnamen benutzt, um sie einfacher von den englischen Keywords und Bibliotheksfunktionen trennen zu können. Im Allgemeinen ist es aber viel sinnvoller, ein Programm vollständig auf Englisch zu schreiben. Schließlich kann man praktisch nie ausschließen, dass nicht zumindest ein Teil eines Programms irgendwann von jemand gelesen wird, der kein Deutsch kann. (Außer das Programm wird sofort unwiederbringlich gelöscht.)

1.2.2 Datentypen und Konstanten

Tabelle 1.1 zeigt einige Datentypen von C.

`char` geht von `(char)-128` bis `(char)127`, `short` mindestens von `(short)-32768` bis `(short)32767`, `int` auch mindestens von `-32768` bis `32767` und `long` von mindestens

Typ	Byte-größe	Beschreibung
`char`	1	ganz kleine Ganzzahl
`short`	≥ 2	kleine Ganzzahl
`int`	≥ 2	Ganzzahl
`long`	≥ 4	große Ganzzahl
`float`	≥ 4	Gleitkommazahl
`double`	≥ 8	genaueres `float`
`void`		der Nicht-Typ

Tabelle 1.1: Wichtige elementare Datentypen von C.

`(long)-2147483648` bis `(long)2147483647`. Gleitkommazahlen sind normalerweise `double`, z. B. `3.1415` oder `0.31415e1`. Beispiele für `float`s sind `(float)3.1415` und `3.1415f`.

Ausserdem gibt es `unsigned` Variationen dieser Typen, z. B. `unsigned char`, die genauso viele Bytes wie die entsprechenden vorzeichenbehafteten Typen brauchen, aber kein Vorzeichen besitzen. Dafür können sie aber mehr postive Werte aufnehmen, z. B. `unsigned char` von 0 bis 255. Hexadezimale Zahlen können mit vorgestelltem `0x` angegeben werde, z. B. `0xff` für 255. Oktalzahlen mit vorgestellter `0`, z. B. `0377` für 255. Schließlich gibt es noch Suffixe analog zu `3.1415f`, die aber eigentlich unnötig sind. (Siehe die Typumwandlungen unten.)

Ein weiteres Beispiel für `int`s sind Zeichen, die mit einfachen Anführungszeichen geschrieben werden, z. B. `'a'` oder `'&'`. Der `int`-Wert ist dann der ASCII-Code dieses Zeichens. `int`s werden außerdem benutzt um Wahrheitswerte (wahr und falsch) zu verarbeiten. Dabei steht `0` für falsch und alles ungleich `0` für wahr.

Die Bytegrößen sind nicht alle festgelegt, sondern hängen vom jeweiligen Compiler ab. Die Idee ist, dass `int` die Größe hat, die der jeweilige Prozessor am besten/schnellsten verarbeiten kann. Die Bytegröße von `short` ist kleiner oder gleich und von `long` größer oder gleich der Größe von `int`. Zur Zeit heißt das meistens, dass `short` 2 Bytes groß ist, `int` und `long` 4 Bytes. Darauf kann man sich aber nicht verlassen. Wenn ein Programm wissen muss, wie groß ein Typ ist, kann es die Länge in Bytes mit `sizeof(`*Typ*`)` erhalten. Der Compiler setzt an dieser Stelle dann die von ihm gewählte Bytegröße ein. (`sizeof(`*Variablenname*`)` und `sizeof` *Variablenname* funktioniert auch und ergibt die Bytegröße des Typs der angegebenen Variable.)

1.2.3 Zuweisungen

... werden als

Variablename `=` *Wert*`;`

geschrieben. Wieder ist der Strichpunkt am kritischsten, aber diesmal markiert er das Ende einer Anweisung.

Zuweisung bedeutet, dass der *Wert* in den Speicherplatz der Variable geschrieben wird. Deswegen ist es wichtig, dass der *Wert* rechts von `=` vom gleichen Typ ist wie die Variable links von `=`, sonst würde der *Wert* eventuell gar nicht in den Speicherplatz der Variable passen. Beispiel:

```
int i;

i = 10;
```

Hier wird also ein Stück Speicherplatz reserviert und dann die Zahl 10 (binär codiert) hineingeschrieben. Das lässt sich auch zusammenfassen als Variablendefinition mit Initialisierung:

```
int i = 10;
```

Welche Zahl stand eigentlich direkt nach der Definition `int i;` noch vor einer Zuweisung in dem Speicherplatz von `i`? Das ist nicht definiert, d.h. es kann irgendwas gewesen sein. Dieser zufällige Wert kann sich auch noch bei jedem Programmablauf ändern, was zu sehr seltsamen Fehlern führen kann. Deswegen initialisieren manche Programmierer jede Variable gleich bei der Definition. Andererseits sind Compiler inzwischen meistens schlau genug, um den Programmierer zu warnen, wenn der (zufällige) Wert einer Variable benutzt wird, bevor die Variable auf einen Wert gesetzt wurde.

1.3 Operatoren

Aus Variablen und Konstanten lassen sich zusammen mit Operatoren und Klammern (`(` und `)`) arithmetische, logische und andere *Ausdrücke* erstellen. (Ein *Ausdruck* mit einem angehängten Strichpunkt ist eine *Anweisung*. *Ausdrücke* dürfen überall stehen wo „Werte“ erwartet werden. *Anweisungen* dagegen können nicht beliebig ineinander geschachtelt, sondern nur hintereinander aufgereiht werden.)

Es werden unäre von binären und ternären Operatoren unterschieden, je nach Anzahl der Werte, die ein Operator verarbeitet. Unäre Operatoren bekommen nur einen Wert, binäre zwei und ternäre drei.

Operator	Beschreibung
`+`	Positives Vorzeichen
`-`	Negatives Vorzeichen
`++`	Inkrementierung
`--`	Dekrementierung
`!`	logisches Komplement
$\sim$	bitweises Komplement
`(`*Typ*`)`	Typumwandlung
`&`	Adressoperator
`*`	Verweisoperator
`sizeof(`*Typ/Wert*`)`	Bytegröße

Tabelle 1.2: Die unären Operatoren.

1.3.1 Unäre Operatoren

Tabelle 1.2 enthält die unären Operatoren. Positives und negatives Vorzeichen sollten klar sein. Im Beispiel

```
int a = 10;
int b;

b = -a;
```

wird `b` auf den Wert `-10` gesetzt.

Das logische Komplement macht aus einer `0` einen Wert ungleich `0` und umgekehrt aus jedem Wert ungleich `0` eine `0`. Das bitweise Komplement invertiert alle Bits einer Zahl.

Die Typumwandlung (engl.: *type cast*) erlaubt es einen Wert eines bestimmten Typs in einen anderen Typ umzuwandeln, wie in dem Beispiel zu Datentypen mit `(float)` schon gezeigt. Es ist auch möglich Gleitkommazahlen in Ganzzahlen (und umgekehrt) zu konvertieren:

```
int i;
double d = 3.1415;

i = (int)d;
```

Da der Datentyp `int` nur ganze Zahlen aufnehmen kann, werden bei der Typumwandlung hier die Nachkommastellen abgeschnitten und `i` bekommt den Wert `3`.

Falls bei einer Zuweisung die Datentypen nicht zusammenpassen, wird automatisch eine Typkonvertierung vorgenommen, deshalb ist die explizite Typumwandlung in diesem Beispiel nicht unbedingt nötig. Die explizite Typumwandlung hat aber den Vorteil, den Leser des Programms daran zu erinnern, dass hier Information verloren geht.

Der Adressoperator `&` braucht als „Wert“ eine Variable, deren Speicheradresse er zurückgibt. (Der Datentyp einer Adresse wird später erklärt.) Umgekehrt braucht der Verweisoperator `*` als Wert eine Speicheradresse und gibt den Wert zurück, der an dieser Adresse im Speicher steht. Einfaches Beispiel:

```
int a = 2000;
int b;

b = *(&a);
```

Mit `int a = 2000;` wird ein Stück Speicher reserviert und der Wert `2000` hineingeschrieben. `(&a)` ergibt die Speicheradresse und mit `*(&a)` wird der Wert an dieser Speicheradresse ausgelesen, also `2000`. Dieser Wert wird mit `b = *(&a)` der Variable `b` zugewiesen. (Also in den Speicher kopiert, der für `b` reserviert wurde.) Das ist eigentlich gar nicht kompliziert, aber die Datentypen von Speicheradressen und Variablen von diesem Typ (sogenannte Zeiger oder Pointer) sind ein Thema für sich, das später behandelt wird.

Der `sizeof`-Operator gibt die Bytegröße eines Datentyps an, z. B. ist `sizeof(char)` gleich 1. Er kann aber auch auf Werte angewendet werden und gibt dann die Größe ihres Datentyps zurück. (`sizeof` selbst gibt einen Wert vom Typ `size_t` zurück, der normalerweise gleich `unsigned int` ist.)

In- und Dekrementierung brauchen nicht einen Wert sondern eine Variable, die sie in- bzw. dekrementieren können, z. B. hat die Variable `a` nach den Zeilen

```
int a = 2000;

++a;
```

den Wert `2001`. (Wie erwähnt ist jeder Ausdruck (hier `++a`) mit nachfolgendem Strichpunkt eine Anweisung.)

Mit Ausnahme des In- und Dekrementoperators können alle unären Operatoren in C nur als Prefix-Operatoren geschrieben werden, das heißt das Symbol für den Operator steht direkt vor dem Wert, auf den der Operator wirkt. Dagegen können der In- und Dekrementoperator sowohl als Prefix- als auch hinter einer Variable als Postfixoperator verwendet werden. Der Unterschied zwischen den beiden Versionen liegt im Rückgabewert. Beim Prefixoperator wird zuerst die Variable erhöht bzw. erniedrigt und der sich ergebende Wert zurückgegeben, d.h. in

```
int a = 2000;
int b;
```

```
b = ++a;
```

bekommt auch `b` den Wert `2001`. `++` und `--` lassen sich aber auch als Postfix-Operatoren verwenden:

```
int a = 2000;
int b;

b = a++;
```

Hier gibt der `++` Operator den ursprünglichen Wert von `a` zurück, deshalb ist `b` dann gleich `2000`. Unabhängig von dem zurückgegebenen Wert erhöht `++` aber den Wert von `a`, deswegen ist `a` wieder gleich `2001`. Entsprechend funktioniert auch der dekrementierende `--` Operator.

Die De- und Inkrementierungsoperatoren sind die einzigen unären Operatoren, die Variablen verändern. Alle anderen unären Operatoren bekommen nur einen Wert und berechnen daraus einen neuen Wert, den sie zurückgeben.

1.3.2 Binäre und ternäre Operatoren

Die binären Operatoren stellen die wichtigsten Rechenoperationen zur Verfügung. Sie sind in Tabelle 1.3 zusammengefasst.

Funktionsaufrufe, Array- und Komponentenzugriffe werden später diskutiert werden.

Die arithmetischen Operatoren funktionieren meistens wie zu erwarten:

```
int a;

a = 1 + 2 * 3;
```

setzt `a` auf den Wert 7.

Vergleiche ergeben `0` (falsch) oder `1` (wahr), die mit den logischen Operatoren weiter verarbeitet werden können, die dann wieder `0` oder `1` ergeben.

Die bitweisen Operatoren verknüpfen jeweils zwei Bits an entsprechenden Stellen. Ein Bitshift `a >> n` entspricht $a/2^n$ (mit Abrundung). Analog bedeutet `a << n` soviel wie $a * 2^n$.

Zuweisungen wurden schon besprochen; ergänzend ist noch zu sagen, dass auch der `=` Operator etwas zurück gibt, nämlich den zugewiesenen Wert. Deshalb ist es möglich zu schreiben

```
int a;
int b;
```

Operator	Beschreibung
`()`	Funktionsaufruf
`[]`	Arrayzugriff
`.`	Komponentenzugriff über eine Strukturvariable
`->`	Komponentenzugriff über eine Strukturadresse
`*`	Multiplikation
`/`	Division
`%`	Modulo (Divisionsrest)
`+`	Addition
`-`	Subtraktion
`>>`	Bitshifts nach rechts
`<<`	Bitshifts nach links
`<`,`>`,`<=`, `>=`	Vergleiche
`==`	Gleichheit
`!=`	Ungleichheit
`&`	bitweises Und
`^`	bitweises exklusives Oder
`|`	bitweises Oder
`&&`	logisches Und
`||`	logisches Oder
`?:`	Bedingungsoperator
`=`	Zuweisung
`+=`, `-=`, `*=`, ...	Zuweisung mit weiterem binären Operator
`,`	Komma

Tabelle 1.3: Die binären und ternären Operatoren.

```
a = (b = 42);
```

und so a und b beide auf den Wert 42 zu setzen.

Die Kombination von Zuweisungen und binären Operatoren erlaubt es z. B. statt

```
int a = 2000;

a = a + 1;
```

zu schreiben

```
int a = 2000;

a += 1;
```

und sich so das doppelte Ausschreiben des Variablennamens zu sparen.

Der einzige ternäre Operator ist der Bedingungsoperator ?:. Ein Beispiel:

```
int a = 42;
int b;

b = (a > 99 ? a - 100 : a);
```

In der letzten Zeile wird zuerst der Teil vor ? ausgewertet. Wenn er wahr ist (ungleich 0), dann wird der Teil zwischen ? und : ausgewertet und zurückgegeben, sonst der Teil nach :. In unserem Fall ist a nicht größer 99, deshalb wird der Wert von a also 42 zurückgegeben.

Mit dem Bedingungsoperator lässt sich schnell das Maximum von zwei Zahlen finden: a > b ? a : b gibt den größeren Wert von a und b zurück.

Das Komma macht nichts anderes, als den linken und dann den rechten Wert auszuwerten, und dann den rechten Wert zurückzugeben.

1.3.3 Automatische Typumwandlungen

... funktionieren meistens so wie man erwartet, d.h. wenn ein Operator zwei Werte unterschiedlichen Datentyps verknüpft, dann wird zunächst der Wert des „kleineren“ bzw. ungenaueren Typs in den „größeren“ bzw. genaueren Datentyp umgewandelt und das Ergebnis in diesem Datentyp berechnet und zurückgegeben

Die genauen Regeln unterscheiden sich aber in den verschiedenen Standards; deshalb ist es sinnvoll im Zweifelsfall immer explizite Typumwandlungen anzugeben.

Operator	Reihenfolge der Auswertung
(),[],.,->	von links
!,∼,++,--,(*Typ*),*,&,sizeof	von rechts
*,/,%	von links
+,-	von links
<<,>>	von links
<,<=,>,>=	von links
==,!=	von links
&	von links
^	von links
|	von links
&&	von links
||	von links
?:	von rechts
=,+=,-=,...	von rechts
,	von links

Tabelle 1.4: Vorrang und Reihenfolge der Auswertung von Operatoren. Operatoren, die weiter oben stehen, haben höheren Vorrang.

Eine Ausnahme wurde schon erwähnt: Der Zuweisungsoperator muss den Wert rechts von = in den Datentyp der Variable links von = konvertieren, wobei eventuell Information verloren geht.

1.3.4 Vorrang von Operatoren und Reihenfolge der Auswertung

Bei Ausdrücken mit mehreren Operatoren ist der Vorrang zwischen den Operatoren wichtig, z. B. ist 1 + 2 * 3 äquivalent zu 1 + (2 * 3) und *nicht* gleich (1 + 2) * 3, da * Vorrang vor + hat. Außerdem kann es unter Umständen wichtig sein, ob zunächst der Wert links vom Operator und dann der Wert rechts ausgewertet wird (Auswertung von links) oder umgekehrt (Auswertung von rechts). In Tabelle 1.4 sind Vorrang und Reihenfolge der Auswertung der C-Operatoren aufgelistet.

Der Vorrang der Operatoren ist sehr wichtig und nicht immer klar. Für die meisten Zweifelsfälle hilft folgende Merkregel: Höchste Priorität haben binäre Operatoren, die üblicherweise ohne Leerzeichen geschrieben werden (a(b), a[b], a.b, a->b), dann kommen unäre Operatoren und dann binäre Operatoren, die mit Leerzeichen geschrieben werden (a + b etc.). (Dazu muss man sich natürlich merken, welche Operatoren ohne Leerzeichen geschrieben werden sollten, aber das

ist etwas einfacher.)

Unter den binären Operatoren mit Leerzeichen kommen zunächst arithmetische Operatoren (mit den üblichen Regeln wie „Punkt vor Strich“), dann Vergleiche, dann bitweise binäre Operatoren und schließlich logische binäre Operatoren. Zuletzt kommt der Bedingungsoperator, die Zuweisungen und das Komma.

In Zweifelsfällen sollten immer Klammern gesetzt werden, um die gewünschte Reihenfolge zu erzwingen und den Code möglichst lesbar zu machen. (Von dem Leser eines C-Programms ist im Allgemeinen nicht zu erwarten, dass er besser C kann als der Autor des Programms.)

1.4 Kontrollstrukturen

1.4.1 Blöcke

... von Anweisungen fangen mit { an und werden mit } beendet. Blöcke können auch geschachtelt werden. Der Programmablauf fängt am Anfang des Blocks an und arbeitet normalerweise alle Anweisungen innerhalb des Blocks der Reihe nach ab.

In C können Variablen entweder außerhalb von Funktionen (*extern*) oder am Anfang eines Blocks innerhalb einer Funktion definiert werden und sind dann nur innerhalb des Blocks gültig, d.h. der Speicherplatz wird am Anfang des Blocks reserviert und, wenn der Programmablauf das Ende des Blocks erreicht, wieder freigegeben.

Diese Variablen heißen deshalb auch *lokale* Variablen, weil ihre *Gültigkeit* auf einen Block beschränkt ist. Das gilt auch, falls die Adresse einer inzwischen ungültigen Variable irgendwie gerettet wurde. Dann kann zwar noch über die Adresse der Wert aus dem Speicherplatz gelesen werden, aber dieser Wert kann bereits von anderen Variablen überschrieben worden sein und hat dann mit dem ehemaligen Wert der ungültigen Variable nichts mehr zu tun.

Externe Variablen dagegen existieren unabhängig vom Programmablauf, denn ihr Speicherplatz wird beim Programmstart reserviert und erst am Programmende wieder freigegeben. Externe Variablen können jederzeit und überall angesprochen werden, deswegen ist es gelegentlich sehr schwer nachvollziehbar, wo und warum der Wert einer externen Variable verändert wurde. Entsprechend ist das Programmieren mit externen Variablen nicht ganz ungefährlich; aufgrund der uneingeschränkten Verfügbarkeit aber sehr verführerisch. (Und dann gibt es noch *globale* Variablen, aber die werden später besprochen.)

Um die Lesbarkeit zu erhöhen sollte das Textlayout die Blöcke durch Einrückungen deutlich machen. Beispiel:

```
{
   int i;

   i = 0;
   {
      int a = 5;

      i = a;
   }
}
```

1.4.2 if-Anweisungen

... haben (in diesem Kurs) diese Syntax:

```
if (Bedingungsausdruck)
{
   Anweisungen
}
```

Die Bedeutung ist ganz einfach: Wenn der *Bedingungsausdruck* wahr ist (also ungleich `0`), dann werden die *Anweisungen* ausgeführt, sonst wird nach dem Block weitergearbeitet. Zum Beispiel:

```
max = a;
if (b > a)
{
   max = b;
}
```

Dies setzt die Variable `max` auf den größeren der beiden Werte von `a` und `b`. Eine Erweiterung von `if` ist der `else`-Block:

```
if (Bedingungswert)
{
   1. Anweisungen
}
else
{
   2. Anweisungen
}
```

Hier werden die *1. Anweisungen* ausgeführt, wenn der *Bedingungsausdruck* ungleich `0` ist und sonst die *2. Anweisungen*. Beispiel:

```
if (b > a)
{
   max = b;
}
else
{
   max = a;
}
```

Das setzt wieder das Maximum, aber diesmal braucht es eine Zuweisung weniger, falls `b > a` ist.

Falls ein Block nach `if` oder `else` nur aus einer Anweisung besteht, können die geschweiften Klammern auch weggelassen werden:

```
if (b > a)
   max = b;
else
   max = a;
```

Das kann aber zu sehr unschönen Fehlern führen und sollte deshalb vermieden werden. Andererseits erlaubt diese Regel auch einen `else if`-Block. In

```
if (a > 100}
{
   a = 100;
}
else
{
   if (a < 0)
   {
      a = 0;
   }
}
```

besteht der `else`-Block nur aus einer `if`-Anweisung, deswegen (und weil Zeilenumbrüche hier keine Rolle spielen) können wir das auch schreiben als

```
if (a > 100}
{
   a = 100;
}
else if (a < 0)
{
   a = 0;
}
```

`if` und bedingte Befehlsausführung allgemein wird gelegentlich als eines der wichtigsten Konzepte von Programmiersprachen angesehen. Allerdings gibt es hier die Gefahr mit `if` zwischen vielen ähnlichen Fällen zu unterscheiden und für jeden Fall eine gesonderte Behandlung zu programmieren. Falls sich der `if`- und der `else`-Block stark ähnlich sehen, ist es im Allgemeinen recht wahrscheinlich, dass die beiden Blöcke zu einem zusammengefasst werden können (eventuell mit kleineren `if-else`-Blöcken).

1.4.3 switch-Anweisungen

Es kommt gelegentlich vor, dass eine Ganzzahl auf Gleichheit mit einer Reihe von Konstanten getestet werden muss. Anstatt viele `if`s zu benutzen, kann zu diesem Zweck auch eine `switch`-Anweisung verwendet werden:

```
switch (Ganzzahl-Ausdruck)
{
   case 1. Konstante:
      1. Anweisungen
   case 2. Konstante:
      2. Anweisungen
   ...
   default:
      default-Anweisungen
}
```

Dabei wird zunächst der *Ganzzahl-Ausdruck* ausgewertet und das Ergebnis mit den *Konstanten* verglichen. Je nach Ausgang der Vergleiche springt der Programmablauf dann zu einer `case`-Marke. Falls das Ergbnis keiner der angegebenen *Konstanten* entspricht, wird zur `default`-Marke gesprungen, falls eine solche angegeben wurde, was nicht unbedingt nötig ist.

Hier ist wichtig, dass der Programmablauf nach dem Sprung alle *Anweisungen* nach der entsprechenden `case`-Marke abarbeitet, auch die *Anweisungen* nach folgenden `case`-oder `default`-Marken! (Diese Marken werden dabei einfach ignoriert.) Deshalb verwendet das nächste Beispiel `break`-Anweisungen, um zum Ende des `switch`-Blocks zu springen. (`break` wird gleich genauer erklärt.)

```
switch (wochentag)
{
   case 0:
      printf("Montag");
      break;
```

```
    case 1:
       printf("Dienstag");
       break;
    case 2:
       printf("Mittwoch");
       break;
    case 3:
       printf("Donnerstag");
       break;
    case 4:
       printf("Freitag");
       break;
    case 5:
       printf("Samstag");
       break;
    case 6:
       printf("Sonntag");
       break;
    default:
       printf("?");
}
```

1.4.4 while-Schleifen

Schleifen sind eine Möglichkeit sich wiederholende Programmteile zusammenzufassen. Als Beispiel soll die Berechnung der Summe der ersten sechs Quadratzahlen dienen (in C wird meistens ab 0 gezählt):

```
int summe;

summe = 0 * 0 + 1 * 1 +
   2 * 2 + 3 * 3 +
   4 * 4 + 5 * 5;
```

Diese Art der Formulierung hat ein paar Mängel: Sie ist unter Umständen sehr aufwendig einzutippen, deswegen passieren leicht Fehler, die auch noch schwer zu finden sind. Außerdem ist sie sehr inflexibel, weil dieser Programmteil offenbar nichts anderes kann als die ersten fünf Quadratzahlen zu addieren. Um zu einer Schleifenformulierung zu kommen, müssen wir erstmal die Berechnung umformulieren:

```
int summe = 0;
int i = 0;

summe = summe + i * i;
i = i + 1;
summe = summe + i * i;
i = i + 1;
summe = summe + i * i;
i = i + 1;
summe = summe + i * i;
i = i + 1;
summe = summe + i * i;
i = i + 1;
```

(Statt `summe = summe + i * i;` könnten wir natürlich auch `summe += i * i;` schreiben; entsprechend `i += 1;` statt `i = i + 1;` oder noch kürzer: `i++;`)

Diese Formulierung hat den Vorteil, dass der sich wiederholende Programmteil sofort ins Auge springt. Mit einer einfachen `while`-Schleife lässt sich die Wiederholung ganz einfach herstellen. Die Syntax der `while`-Schleife ist:

```
while(Bedingungsausdruck)
{
   Anweisungen
}
```

und bedeutet, dass, falls der *Bedingungsausdruck* ungleich `0` (also wahr) ist, die *Anweisungen* ausgeführt werden. Danach springt der Programmablauf wieder zurück und berechnet den *Bedingungsausdruck* neu. (Das ist nötig, weil sich Variablenwerte inzwischen geändert haben können.) Falls der neue Wert des *Bedingungsausdruck* wieder ungleich `0` ist, werden die Anweisungen nochmal ausgeführt. Das geht solange, bis der *Bedingungsausdruck* gleich `0` (falsch) ist; dann springt der Programmablauf an das Ende des Anweisungsblocks und macht da weiter.

Damit sieht unsere Berechnung der Summe der ersten fünf Quadratzahlen so aus:

```
int summe = 0;
int i = 0;

while (i <= 5)
{
   summe = summe + i * i;
   i = i + 1;
}
```

Das sieht nicht so aus, als ob es sehr viel kürzer wäre als unsere erste Version, *aber* wir können hiermit auch ebenso einfach die Summe der ersten 100 Quadratzahlen berechnen. Wenn wir die `5` durch eine Variable ersetzen, können wir sogar die Summe von einer Anzahl

von Quadratzahlen berechnen, die wir noch gar nicht kennen, wenn wir das Programm schreiben, sondern die erst beim Programmablauf bekannt ist.

1.4.5 do-Schleifen

... gehorchen der Syntax

```
do
{
   Anweisungen
}
while (Bedingungsausdruck);
```

Die Bedeutung ist ähnlich wie bei `while`, allerdings werden die *Anweisungen* mindestens einmal ausgeführt. Danach wird der *Bedingungsausdruck* ausgewertet und, falls er ungleich `0` (also wahr) ist, die Schleife wiederholt. `do` wird verhältnismäßig selten genutzt.

1.4.6 for-Schleifen

... folgen der Syntax

```
for (Startausdruck;Bedingungsausdruck
;
Iterationsausdruck)
{
   Anweisungen
}
```

(normalerweise ist der `for(...)`-Teil in einer Zeile) und sind äquivalent zu

```
Startausdruck;
while (Bedingungsausdruck)
{
   Anweisungen
   Iterationsausdruck;
}
```

Das ist eigentlich alles, was es zu `for` zu sagen gibt. Vielleicht noch ein Beispiel, statt

```
x = 0;
while (x < 400)
{
   y = 0;
   while (y < 300)
   {
      glColor3f(x / 300.0,
         y / 400.0, 1.0);
      glVertex2i(x, y);
      y = y + 1;
   }
   x = x + 1;
}
```

können wir jetzt kürzer schreiben:

```
for (x = 0; x < 400; x++)
{
   for (y = 0; y < 300; y++)
   {
      glColor3f(x / 300.0,
         y / 400.0, 1.0);
      glVertex2i(x, y);
   }
}
```

wobei `x = x + 1` durch `x++` und `y = y + 1` durch `y++` ersetzt wurde. Die Formulierung mit `for` ist zumindest kompakter und hat sich so sehr eingebürgert, dass sie für erfahrene C-Programmierer auch etwas leichter lesbar ist.

Anfänglich ist die `for`-Schleife aber vielleicht etwas schwieriger zu verstehen, vor allem weil nicht deutlich wird, ob die Schleife mindestens einmal durchlaufen wird. (Wird sie nicht!) Das ist auch der wesentliche Unterschied zwischen `while` und `for` auf der einen Seite und der `do`-Schleife auf der anderen Seite.

1.4.7 break-Anweisungen

... können nicht nur hinter das Ende einer `switch`-Anweisung springen, sondern springen allgemein hinter das Ende der innersten umgebenden `while`-, `for`-, `do`-Schleife oder eben `switch`-Anweisung, je nach dem wo die `break`-Anweisung steht. Dabei wird die Ausführung der entsprechenden Schleife abgebrochen. Zum Beispiel:

```
x = 0.5;

for (i = 0; i < 1000; i++)
{
   x =  a * x * (1.0 - x);
   if (x > 1.0)
   {
      break;
   }
}
```

Hier wird 1000mal die Iterationsvorschrift $x \leftarrow a \cdot x \cdot (1-x)$ mit Startwert 0.5 angewendet. Wenn aber `x` irgendwann größer als `1.0` ist, wird diese Schleife durch die `break`-Anweisung beendet. Das ist fast gleichbedeutend mit

```
x = 0.5;

for (i = 0;
   i < 1000 && x <= 1.0;
   i++)
{
   x =  a * x * (1.0 - x);
}
```

Der Unterschied ist nicht ganz einfach zu sehen (mit `while` statt `for` wäre er klarer). Er besteht darin, dass falls `x` jemals größer als `1.0` wird, im zweiten Fall einmal mehr ein `i++` ausgeführt wird. (Außerdem ergibt sich für einen Startwert von `x` größer als `1.0` ein unterschiedliches Verhalten .)

1.4.8 continue-Anweisungen

... springen ähnlich wie `break` an das Ende der innersten umgebenden `while`-, `for`- oder `do`-Schleife, aber fahren dann in der normalen Schleifenabarbeitung fort anstatt die Schleife abzubrechen. In der `for`-Schleife wird auch die Iterationsanweisung ausgeführt. `continue`-Anweisungen werden etwas seltener als `break`-Anweisungen eingesetzt.

1.4.9 goto-Anweisungen und Marken

Mit `goto` kann der Programmablauf vor einer beliebigen Anweisung der gleichen Funktion fortgesetzt werden. Dazu muss vor dieser Anweisung eine Marke gesetzt werden, deren Name denselben Einschränkungen wie Variablennamen unterliegt. Marken selbst werden durch einen nachfolgenden Doppelpunkt markiert. Im Beispiel

```
for (x = 0; x < 400; x++)
{
   for (y = 0; y < 300; y++)
   {
      ...
      if (fehler)
      {
         goto Fehlerbehandlung;
      }
   }
   ...

Fehlerbehandlung:
   ...
```

werden die beiden geschachtelten Schleifen durch einen Sprung zur Marke `Fehlerbehandlung` beendet.

Es gibt wenige Gründe `goto` zu verwenden, aber viele es zu lassen. (Stichwort: Spaghetti-Code durch verschlungene Sprünge.) Das Beispiel zeigt zwei Ausnahmen: Reaktion auf Fehlersituationen und das Abbrechen von verschachtelten Schleifen, was mit einer `break`-Anweisung nicht möglich ist, da `break` nur die innerste Schleife abbricht.

1.5 Funktionen

1.5.1 Funktionsdefinitionen

Funktionen werden meistens nach dem Schema

Rückgabetyp Funktionsname`(` *Argumenttyp Argumentname* `,` ... `)`
{
   ...
}

definiert (und zwar immer *extern*, also nicht innerhalb von anderen Funktionen oder Blöcken). Zum Beispiel:

```
int verdopple(int eingabe)
{
   int ausgabe;

   ausgabe = 2 * eingabe;

   return(ausgabe);
}
```

Das ist so zu lesen: Hier wird eine Funktion namens `verdopple` definiert, die einen Wert vom Typ `int` zurückgibt und ein Argument vom Typ `int` erwartet. Innerhalb der Funktionsdefinition ist eine Variable namens `eingabe` definiert, die bei einem Funktionsaufruf mit dem Argumentwert initialisiert wird. Außerdem wird eine weitere Variable vom Typ `int` namens `ausgabe` definiert, der der verdoppelte Wert des

Arguments zugewiesen wird. Mit Hilfe der `return`-Anweisung wird dieser verdoppelte Wert dann zurückgegeben.

Die in der Funktion definierten lokalen Variablen (einschließlich der „Argumentvariablen“) verlieren ihre Gültigkeit (ihre Speicherplatzreservierung), sobald der Programmablauf das Ende der Funktion erreicht. Zwei wichtige Punkte dazu:

Erstens: Innerhalb einer Funktion sind nur die dort definierten, lokalen Variablen und die im Modul definierten externen Variablen *sichtbar*, können also benutzt werden. Alle lokalen Variablen anderer Funktionen können nicht über ihren Namen benutzt werden. Deshalb müssen alle Informationen, die die Funktion benötigt, entweder über externe Variablen (nur in Ausnahmefällen empfehlenswert!) oder als Argumente übergeben werden.

Zweitens: *Immer* wenn der Programmablauf an den Anfang der Funktion kommt, wird *neuer* Speicherplatz für die lokalen Variablen der Funktion angelegt. Und das auch wenn schon einmal für diese Variablen Speicherplatz reserviert wurde, der noch nicht wieder freigegeben wurde! In diesem Fall existieren dann mehrere Variablen (an verschiedenen Positionen im Speicher) mit gleichen Namen. Aufgrund der Sichtbarkeitsregel ist aber immer nur genau eine davon ansprechbar.

Durch das Voranstellen des Modifizierers `static` (Bsp. `static int a;`) kann verhindert werden, dass eine lokale Variable bei jedem Funktionsaufruf neu erzeugt wird. Diese sogenannten *statischen* Variablen behalten ihren Wert auch zwischen Funktionsaufrufen und können z.B. Werte aus vorhergehenden Aufrufen speichern, ohne extern definierte Variablen verwenden zu müssen. Die Verwendung von `static` für externe Variablen und für Funktionen hat eine ganz andere Bedeutung. Diese wird allerdings erst in Kapitel 4.2.3 beprochen.

Falls eine Funktion keine Argumente braucht, muss als Argumenttyp `void` (ohne Argumentname) angegeben werden. Falls sie keinen Rückgabewert hat, muss `void` als Rückgabetyp angegeben werden. Zum Beispiel:

```
void tue_nichts(void)
{
   return;
}
```

Die `return`-Anweisung am Blockende ist dabei nicht notwendig, aber auch nicht verboten.

1.5.2 Funktionsaufrufe

Nach dieser Funktionsdefinition könnte an anderer Stelle im Programm ein Funktionsaufruf stehen, z. B. so

```
int d;

d = verdopple(100);
```

Bei dem Aufruf von `verdopple(100)` wird zunächst das Argument ausgewertet, was hier einfach ist, weil es gerade `100` ist. Jetzt erfolgt ein Sprung in die Funktion. Hier wird eine Variable namens `eingabe` definiert (Speicherplatz reserviert) und auf den Argumentwert `100` gesetzt. Dann wird eine Variable `ausgabe` definiert. Anschließend wird der Wert der Variable `eingabe` mit `2` multipliziert und das Ergebnis in der Variable `ausgabe` gespeichert. Die `return`-Anweisung liest diesen Wert aus der Variable `ausgabe` und springt damit wieder zurück in die Zeile `d = verdopple(100);`. Hier wird der Rückgabewert `200` dann in der Variable `d` gespeichert.

Allgemein sieht ein Funktionsaufruf so aus:

Funktionsname`(`*Argument*`,...)`

Funktionsaufrufe sind ganz normale Ausdrücke und können deshalb durch Anhängen eines Strichpunkts zu Anweisungen gemacht werden. Die Klammern in einem Funktionsaufruf können mit etwas Phantasie als binärer Operator aufgefasst werden, dessen linker Wert der *Funktionsname* ist und dessen rechter Wert die Liste der *Argumente* ist.

Wichtig ist, dass die Klammern auch gesetzt werden, wenn die Funktion gar keine Argumente erwartet. Die Klammern bleiben dann einfach leer und kennzeichnen, dass es sich um einen Funktionsaufruf handelt. (Was der Funktionsname alleine bedeutet, wird später erklärt.)

1.5.3 Funktionsdeklarationen

Falls der Funktionsaufruf vor der Funktionsdefinition steht, bekommt der Compiler beim Übersetzen des Programms ein Problem, weil er einen Funktionsaufruf übersetzen soll, ohne die Funktion zu kennen. Um trotzdem solche Funktionsaufrufe zu ermöglichen, muss die Funktion dem Compiler bekannt gemacht, sprich deklariert, werden. Dazu dient der Funktionsprototyp, der nur aus dem Funktionskopf der Funktionsdefinition (alles außer dem Anweisungsblock) gefolgt von einem Strichpunkt besteht. In unserem Beispiel wäre das:

```
int verdopple(int eingabe);
```

Das gleiche Problem tritt auch bei Bibliotheksfunktionen auf: Jede Bibliotheksfunktion, die benutzt werden soll, muss vorher durch einen Funktionsprototypen bekannt gemacht werden. Deshalb werden Funktionsprototypen (mit anderen Deklarationen) in sogenannten Header- oder `.h`-Dateien zusammengefasst, damit sie mittels der `#include`-Anweisung in ein C-Programm aufgenommen werden können. Zum Beispiel gibt es eine Standard-Headerdatei namens `stdio.h` in der unter anderem die wichtige Funktion `printf` zur formatierten Ausgabe deklariert wird. Wenn `printf` verwendet werden soll, muss deshalb im Programm (normalerweise am Anfang) eine Zeile

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

stehen. Wichtig dabei ist, dass das # am Zeilenanfang stehen sollte. (Es ist hier nur eingrückt, um es besser vom Haupttext zu trennen.)

1.6 Beispielprogramme

Am Ende dieses Kapitels werden noch einige Beispielprogramme besprochen, die alle mit den bisher vorgestellten Elementen der Sprache C realisiert werden können. Außerdem werden noch einige nützliche Bibliotheksfunktionen eingeführt.

1.6.1 Einfache Textverarbeitung

In Kapitel 1.1.1 wurde bereits die Funktion `printf` zur Ausgabe von Zeichenketten auf der Standardausgabe vorgestellt. Sie ist ein Bestandteil der C-Standardbibliothek und wird in der Header-Datei `stdio.h` deklariert. Im Folgenden wird diese Funktion genauer erläutert und weitere hilfreiche Funktionen zur Ein- und Ausgabe von Text vorgestellt.

Aus- und Eingabe einzelner Zeichen

Die Funktion `putchar()` gibt ein einzelnes Zeichen auf der Standardausgabe aus. Beispielsweise erscheint mit

```
int c = 'a';

putchar(c);
```

das Zeichen a auf dem Bildschirm.

Mit `getchar()` wird ein einzelnes Zeichen (immer das nächste im Eingabestrom) von der Standardeingabe eingelesen. Nach der Ausführung der folgenden Zeilen enthält die Variable c also das nächste Eingabezeichen.

```
int c;

c = getchar();
```

Mit diesen zwei einfachen Funktionen können bereits sehr viele interessante Programme zur Verarbeitung von Ein- und Ausgabeströmen realisiert werden. Das folgende Beispielprogramm kopiert z.B. die Standardeingabe Zeichen für Zeichen auf die Standardausgabe.

```
#include <stdio.h>

int main(void)
{
   int c;

   c = getchar();
   while (c != EOF)
   {
      putchar(c);
      c = getchar();
   }

return(0);
   }
```

`EOF` ist eine Konstante und wird von getchar() zurückgegeben, wenn das Ende einer Datei (*end of file*) bzw. des Eingabestroms erreicht wurde. Hierdurch wird auch klar warum die Variable `c` nicht vom Typ `char` sonder vom Typ `int` ist. `c` muss nämlich groß genug sein, damit es nicht nur alle möglichen Zeichen speichern kann, sondern auch den Wert `EOF`.

Da die Standardausgabe in der Regel gepuffert ist, werden durch obiges Beispielprogramm die eingegebenen Zeichen normalerweise nicht sofort nach der Eingabe ausgegeben, sondern z.B. erst beim auftreten eines Zeilenumbruchs. Das Ende einer Datei (`EOF`) kann auf der Standardeingabe unter Unixsystemen übrigens durch die Tastenkombination `STRG + D` simuliert werden.

Da unter Unixsystemen mit Hilfe des <- und des >-Zeichens der Inhalt einer Datei auf die Standardeingabe bzw. die Standardausgabe auf eine Datei umglenkt weren kann, können mit dem obigen Programm auch Da-

teien ausgegeben bzw. Texte in eine Datei geschrieben werden.

Wenn das ausführbare Programm z.B. den Namen `copyChars` hat, kann man mit

```
copyChars <test
```

den Inhalt der Datei `test` auf dem Bildschirm ausgeben. Mit

```
copyChars <quelle >ziel
```

wird der Inhalt der Datei `quelle` in die Datei `ziel` geschrieben.

Formatierte Textausgabe

Mit der Funktion `printf` können auch Werte von Variablen ausgegeben werden, was sehr praktisch ist, um nach Fehlern in Programmen zu suchen. Dazu muss in dem ersten Argument von `printf` (dem Formatstring) für jede Variable ein *Formatelement* angegeben werden, z. B. `%d` für `int`s, `%ld`, für `long`s, `%g` für `float`s und `double`s und `%s` für Zeichenketten (Strings oder `char`-Arrays, werden später genauer behandelt). Die Werte der Variablen müssen als weitere Argumente an `printf` übergeben werden und zwar in der Reihenfolge, wie sie durch die Formatelemente bestimmt ist. Zum Beispiel erzeugt das Programm

```
#include <stdio.h>

int main(void)
{
   int i = 42;
   long o = 1000000000;
   double x = 3.1415;

   printf("i=%d, o=%ld, d=%g\n",
      i, o, x);

   printf("dies %s ein string\n",
      "ist");

   return(0);
}
```

die Ausgabe

```
i=42, o=1000000000, d=3.1415
dies ist ein string
```

Escapesequenz	Bedeutung
`\a`	Gongzeichen (Bell)
`\b`	Backspace
`\f`	Seitenvorschub (Formfeed)
`\n`	Zeilenendezeichen (Newline)
`\r`	Zeilenrücklauf (Carriage Return)
`\t`	horizontaler Tabulator
`\v`	vertikaler Tabulator
`\\`	das Backslashzeichen selbst
`\'`	Apostroph
`\"`	Anführungszeichen
`\?`	Fragezeichen
`\ooo`	numerische Angabe eines Zeichens in Oktal
`\xhh`	numerische Angabe eines Zeichens in Hexadezimal

Tabelle 1.5: Escapesequenzen

Um in einer Zeichenkette auch spezielle Kontrollzeichen, z.B. das Ende einer Zeile, darstellen zu können, muss man sogenannte Escapesequenzen verwenden. Diese werden durch zwei Zeichen repräsentiert, wobei das einleitende Zeichen ein Backslash (\) ist. Ein Beispiel für solch eine Escapesequenz, das Zeilenendezeichn `\n` taucht bereits in obigen Beispiel auf. Mit `\t` wird ein Tabulatorzeichen gesetzt. In Tabelle 1.5, werden alle möglichen Escapsequenzen aufgeführt.

Bei Zeichenketten tritt das Problem auf, dass das Zeilenende doch wichtig ist, denn innerhalb einer in Anführungszeichen gesetzten Zeichenkette darf kein Zeilenumbruch stattfinden. Die Lösung ist, dass Zeichenketten hintereinander zu einer zusammengefasst werden, also `"Halli" "hallo"` das gleiche wie `"Hallihallo"` ist. Wenn eine Zeichenkette so aufgeteilt wird, darf aber auch ein Zeilenumbruch zwischen den Teilen stehen, z. B.:

```
#include <stdio.h>

int main(void)
{
   int i = 42;

   printf("Hier kommt"
      "ein int: %d und"
      "ein double: %g\n",
      i, 3.1415);
```

```
    return 0;
}
```

Formatierte Texteingabe

Mit `getchar()` kann man einzelne Zeichen von der Standardeingabe einlesen. Für Programme bei denen die Standardeingabe für die Interaktion mit dem Anwender genutzt wird, z.B. um Zahlenwerte für Berechnungen einzulesen, ist die Verwendung dieser Funktion jedoch relativ aufwendig.

Deshalb stellt die Standardbibliothek von C auch Funktionen zur Texteingabe, bei denen die Texteingabe direkt in eine oder mehrere Variablen vom gewünschten Typ eingelesen wird. Eine dieser Funktionen ist die `scanf`. Die Verwendung von `scanf` ähnelt weitgehend der von `printf`, mit der Ausnahme das nicht die Werte der übergebenen Variablen ausgegeben werden sondern ihr Inhalt entsprechend belegt wird. Wie `printf` verwendet auch `scanf` einen Formatstrings der die Struktur der Eingabe beschreibt, das heißt er legt über Formatelemente fest wo innerhalb des Eingabestrings Werte stehen die den Variablen zugewiesen werden sollen. Dabei werden dieselben Formatelemente verwendet, wie sie schon im vorherigen Abschnitt beschrieben wurden.

Natürlich kann es vorkommen, dass der Benutzer eine fehlerhafte Eingabe tätigt, z.B. eine Fließkommazahl eingibt, wenn eine Ganzzahl erwartet wird. Deshalb liefert `scanf` als Ergebnis die Anzahl der erfolgreich zugewiesenen Eingabelemente zurück.

Ein einfaches Programm, das vom Benutzer die Eingabe einer ganzen Zahl erwartet und das Quadrat dieser Zahl als Ausgabe zurückliefert, könnte folgendermaßen aussehen:

```
#include <stdio.h>

int main(void)
{
   int i;

   printf("Bitte eine Ganzahl"
          "eingeben: ");
   if (scanf("%d", &i) == 1) {
       printf("Das Quadrat von %d"
              "ist %d\n", i, i*i);
   } else {
       printf("Die Eingabe war"
              "fehlerhaft\n");
   }

   return 0;
}
```

1.6.2 Mathematische Funktionen

C selbst enthält nur Grundrechenarten, aber es gibt eine mathematische C-Standardbibliothek, die viele nützliche Funktionen enthält. Folgende Funktionen bekommen als Argument alle ein `double` und geben ein `double` zurück: `sin`, `asin` (Arcussinus), `cos`, `acos`, `tan`, `atan`, `sinh`, `cosh`, `tanh`, `exp`, `log`, `sqrt` (Wurzel), `fabs` (Betrag), `floor` (abgerundet), `ceil` (aufgerundet). Für zwei `double`s `a` und `b` berechnet `pow(a,b)` a^b und `fmod(a,b)` den Divisionsrest von `a` / `b`.

Um diese Funktionen zu benutzen, muss die Headerdatei `math.h` `#include`d werden:

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

int main(void)
{
   double x = 0.1;

   printf("sin(%g)= %g\n",
      x, sin(x));

   return 0;
}
```

Eine Schwierigkeit ergibt sich bei der Übersetzung von Programmen die Funktionen aus `math.h` verwenden. Hier muss dem C-Compiler nämlich mitgeteilt werden, dass die Mathematik-Bibliothek verwendet wird. Dies geschieht meistens mit der Kommandozeilenoption `-lm`, also heißt der komplette Aufruf zum Compilieren dann:

```
gcc main.c -lm
```

1.6.3 Beispiel: Fakultätsberechnung

Als Beispiel für ein paar einfache Programme folgen verschiedene Berechnungsmöglichkeiten für die Fakultät von natürlichen Zahlen $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$, wobei $0! = 1$ definiert wird. Zum Beispiel ist $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, $10! = 3628800$, usw.

Iterative Berechnung

So wie die Fakultät gerade definiert wurde, lässt sie sich auch mit einer Schleife (iterativ) berechnen. Zum Beispiel für $n = 10$:

```
int n = 10;
int i = 1;
long ergebnis = 1;

while (i <= n)
{
   ergebnis = ergebnis * i;
   i = i + 1;
}
```

Diese Schleife zählt `i` von `1` bis `11`. Die Anweisungen in der Schleife werden aber nur für die Werte `1` bis `10` ausgeführt. Dabei wird `i` auf das Ergebnis multipliziert und dann `i` weitergezählt.

Hier wurde für `n` und `i` der Datentyp `int` benutzt; weil aber die Fakultät sehr schnell wächst, ist die Variable `ergebnis` vom Typ `long`. Die automatische Typkonvertierung stellt sicher, dass `i` bei der Multiplikation zuerst in ein `long` umgewandelt wird, so dass es keine Probleme mit den Datentypen gibt.

Diese Schleife sollte besser als Funktion geschrieben werden:

```
long fakultaet(int n)
{
   int i = 1;
   long ergebnis = 1;

   while (i <= n)
   {
      ergebnis = ergebnis * i;
      i = i + 1;
   }

   return(ergebnis);
}
```

Dann sieht das ganze Programm zum Beispiel so aus:

```
#include <stdio.h>

long fakultaet(int n)
{
   int i = 1;
   long ergebnis = 1;

   while (i <= n)
   {
      ergebnis = ergebnis * i;
      i = i + 1;
   }

   return(ergebnis);
}

int main(void)
{
   printf("10! = %ld\n",
      fakultaet(10));

   return 0;
}
```

Das ist zwar ein ganzes Programm, aber noch lange nicht komplett. Hier mal ein Beispiel wie ein komplettes, auf Englisch geschriebenes und kommentiertes Programm aussehen könnte:

```
/*
 * main.c
 *
 * Example for the calculation of
 * the factorial 10!
 *
 * written by Martin Kraus
 * (Martin_Kraus_Germany@yahoo.com)
 *
 * Revision History
 * ----------------
 * 07/21/2000  mk  created
 *
 */

/* includes */

#include <stdio.h>


/* function prototypes */

long factorial(int n);


/* function definitions */

int main(void)
   /* prints the factorial 10!. */
```

```
{
   printf("10! = %ld\n",
      factorial(10));

   return 0;
}

long factorial(int n)
   /* returns the factorial of n. */
{
   int i = 1;
   long result = 1;

   while (i <= n)
   {
      result = result * i;
      i = i + 1;
   }

   return(result);
}
```

Beim ersten Lesen lenken die Kommentare in diesem kleinen Beispiel wahrscheinlich eher vom Inhalt ab. Bei größeren Projekten mit standardisierten Kommentaren erleichtern sie die Orientierung aber unheimlich — auch bei Dateien die nicht größer sind als diese hier.

Rekursive Berechnung

Es gibt noch einen ganz anderen Weg, die Fakultät zu berechnen, nämlich nach der rekursiven Formel $n! = n \cdot (n-1)!$ zusammen mit der Definition $0! = 1$. Dabei wird ausgenutzt, dass $n!$ leicht berechnet werden kann, wenn $(n-1)!$ bekannt ist. Entsprechend kann $(n-1)!$ aus $(n-2)!$ berechnet werden, usw. Das darf aber nicht endlos weiter getrieben werden, sondern muss irgendwann aufhöhren, wozu die Definition $0! = 1$ gut ist.

Unsere `fakultaet()`-Funktion kann dann so geschrieben werden:

```
long fakultaet(int n)
{
   int ergebnis;

   if (n == 0)
   {
      ergebnis = 1;
   }
   else
   {
      ergebnis = n *
         fakultaet(n - 1);
   }
   return(ergebnis);
}
```

Weil sich die Funktion `fakultaet()` selbst aufruft, nennt man das auch direkte Rekursion.

Bei rekursiven Funktionen wird das Konzept der lokalen Variablen besonders wichtig: Bei jedem Aufruf von `fakultaet()` werden neue Variablen namens `n` und `ergebnis` definiert, also neuer Speicherplatz reserviert. Zum Beispiel wird bei dem Funktionsaufruf `fakultaet(2)` eine Variable `n` mit Wert `2` und eine Variable `ergebnis` definiert. In dieser Funktion wird `fakultaet(n - 1)`, also `fakultaet(1)` aufgerufen. In *diesem* Funktionsaufruf wird eine *neue* Variable `n` mit Wert `1` und eine neue Variable `ergebnis` definiert. Dann wird `fakultaet(0)` aufgerufen und wieder eine neue Variable `n` diesmal mit Wert `0` definiert und eine neue Variable `ergebnis`, die auf `1` gesetzt wird. In diesem Moment gibt es also jeweils 3 Variablen namens `n` und `ergebnis`, wobei alle `n`s unterschiedliche Werte haben und nur ein `ergebnis` bisher einen definierten Wert besitzt. Wenn der Programmablauf dann zurückspringt verlieren die jeweils sichtbaren Variablen ihre Gültigkeit und die `ergebnis` Variablen werden auf definierte Werte gesetzt.

Berechnung mit Gleitkommazahlen

Leider sind unsere bisherigen Definitionen von `fakultaet()` bestenfalls akademische Beispiele. Das liegt daran, dass schon $13! = 6227020800$ eine größere Zahl ist als mit einem 4-Byte `long` verarbeitet werden kann. Mit Gleitkommazahlen kommen wir etwas weiter:

```
#include <stdio.h>

double fakultaet(int n)
{
   int i = 1;
   double ergebnis = 1.0;

   while (i <= n)
   {
      ergebnis = ergebnis * i;
      i = i + 1;
   }
```

```
   return(ergebnis);
}

int main(void)
{
   printf("13! = %g\n",
      fakultaet(13));

   return 0;
}
```

Aber auch der Exponent von Gleitkommazahlen unterliegt gewissen Einschränkungen, so dass wir bei einem 8-Byte-`double` typischerweise nur bis etwa $170! = 7.25741\ldots\cdot 10^{306}$ kommen. (Ein Ausweg wäre die Verwendung des in diesem Kurs nicht behandelten Typs `long double` statt `double`, falls das etwas hilft. Viel weiter kommt man aber auch damit nicht.)

Man könnte der Meinung sein, dass es keinen Sinn macht größere Fakultäten zu berechnen, wenn sie sowieso nicht mit `double`s dargestellt werden können. Das ist aber nicht ganz richtig. Dazu genügt ein kurzer Blick auf die Anwendungen für Fakultäten. Unter anderem werden sie intensiv in der Stochastik verwendet. Sie tauchen zum Beispiel in Binomialkoeffizienten auf.

Binomialkoeffizienten

Der Binomialkoeffizient $\binom{n}{k}$, sprich: „n über k“ oder „k aus n“, ist definiert als $\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!}$. Er ist sehr nützlich, vor allem in der binomischen Formel von der er seinen Namen hat:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

Außerdem gibt er zum Beispiel an, wieviele Möglichkeiten es gibt, k verschiedene Elemente aus einer Menge von n unterscheidbaren Elementen auszuwählen.

Die Berechnung mit Hilfe der `double`-Variante von `fakultaet()` ist einfach:

```
double binomial(int n, int k)
{
   return(fakultaet(n) /
      fakultaet(k) /
      fakultaet(n - k));
}
```

`binomial(170, 3)` ergibt zum Beispiel `804440.0`. Aber $\binom{200}{3}$ ist 1313400 und kann mit 8-Byte-`double`s schon nicht mehr so ausgerechnet werden, weil die Fakultäten nicht berechnet werden können.

In der Praxis wollen Anwender aber natürlich doch solche Binomialkoeffizienten (oder allgemein: Verhältnisse von großen Fakultäten) berechnen. Eine sehr effiziente Möglichkeit dazu, soll im nächsten Abschnitt kurz vorgestellt werden.

Stirlingsche Formel

... ist eine Näherung für den natürlichen Logarithmus einer Fakultät $\ln n!$:

$$\ln n! \approx \left(n+\frac{1}{2}\right)\ln n - n + \frac{1}{2}\ln(2\pi)$$

Oder in C:

```
#include <math.h>

double fakultaet_ln(int n)
{
   return((n + 0.5) * log(n) - n +
      0.91893853320467274178);
}
```

Wieder führt die automatische Typkonvertierung (z. B. von `n` in `n + 0.5`) zum erwarteten Ergebnis. Weil `math.h` benutzt wird, muss beim Compilieren wieder mit `-lm` die Mathematikbibliothek angegeben werden.

Damit können wir die Funktion `binomial()` umschreiben, denn

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k!\cdot(n-k)!} \\ &= \frac{\exp(\ln n!)}{\exp(\ln k!)\cdot\exp(\ln(n-k)!)} \\ &= \exp(\ln n! - \ln k! - \ln(n-k)!)\end{aligned}$$

Oder in C:

```
#include <math.h>

double binomial(int n, int k)
{
   return(exp(fakultaet_ln(n) -
      fakultaet(k) -
      fakultaet(n-k)));
}
```

Damit ergibt `binomial(200, 3)` dann `1.35027e+06`, was zumindest ein Näherung für 1313400 ist. Hier nocheinmal das ganze Programm dazu:

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

double fakultaet_ln(int n)
{
   return((n + 0.5) * log(n) - n +
      0.91893853320467274178);
}

double binomial(int n, int k)
{
   return(exp(fakultaet_ln(n)-
      fakultaet_ln(k) -
      fakultaet_ln(n - k)));
}

int main(void)
{
   printf("(200 ueber 3) = %g\n",
      binomial(200, 3));

   return 0;
}
```

Übungen zum ersten Tag

Aufgabe I.1

Schreiben Sie ein C-Programm das den Flächeninhalt und den Umfang eines Kreises berechnet. Dabei soll der Radius vom Anwender interaktiv eingegeben werden. Bei einer fehlerhaften Eingabe, soll der Anwender eine Fehlermeldung erhalten. Ansonsten sollen die beiden errechneten Werte mit einer Beschreibung ausgegeben werden. Für die Kreiszahl π können Sie den Näherungsert 3.1415 verwenden.

Aufgabe I.2

Schreiben Sie ein C-Programm, das Zeichen für Zeichen von der Standardeingabe einliest und dabei die Anzahl aller auftretenden Zeichen, aller ganzen Wörter und aller Zeilenumbrüche zählt und die jeweiligen Anzahlen nach dem Ende der Eingabe bzw. am Dateiende ausgibt. Wörter sind dabei durch Leerzeichen, Tabulatorzeichen oder Zeilenumbrüche getrennte Zeichenfolgen. Testen Sie ihr Programm, indem Sie die C-Datei des Programms selbst auf die Standardeingabe umlenken.

Aufgabe I.3

In der folgenden Aufgabe soll eine iterative Berechnung durchgeführt werden, bei der ein neuer Wert aus dem jeweils vorhergehenden mit Hilfe einer Iterationsvorschrift berechnet wird.

Gegeben sei $f_0 = 0.5$ und die Berechnungsvorschrift $f_{n+1} = a \cdot f_n \cdot (1 - f_n)$.

Schreiben Sie ein C-Programm, dass für $a = 2.0, 2.1, 2.2, \ldots, 3.8, 3.9$ die Werte f_{100}, f_{101}, f_{102}, f_{103} und f_{104} berechnet und zusammen mit a ausgibt. Erweitern Sie dann Ihr Programm so, dass der minimal und maximal Wert für n für den Werte ausgegeben werden sollen vom Benutzer interaktiv eingegeben werden kann.

Aufgabe I.4

Schreiben Sie einen einfachen Taschenrechner, der die vier Grundrechenarten beherrscht. Der Anwender soll dabei die Rechenvorschrift in der Form `ZAHL OPERATOR ZAHL` über die Standardeingabe eingeben. `ZAHL` ist dabei eine beliebige Gleitkommazahl und `OPERATOR` ein Zeichen aus der Menge $\{+, -, *, /\}$. Die Eingabe soll auf Korrekheit überprüft werden und bei Bedarf eine aussagekräftige Fehlermeldung ausgegeben werden. Nach Ausgabe des Ergebnisses bzw. der Fehlermeldung soll der Benutzer entscheiden können, ob er das Programm fortsetzen will.

Teil II

Zweiter Tag

Kapitel 2

Datenstrukturen in C

2.1 Zeiger

Zeiger sind Variablen, die Adressen speichern. Die Werte von Zeigern sind also Adressen. Das hört sich einfach an und ist es auch. Gefährlich ist es deswegen, weil an den Adressen irgendetwas stehen kann. Denn mit Hilfe von Zeigern ist es möglich, beliebige Speicherinhalte zu ändern, z. B. auch Speicherplätze, die die Maschinenbefehle eines übersetzten Programms beinhalten! Werden Maschinenbefehle willkürlich überschrieben und das so veränderte Maschinenprogramm ausgeführt, führt das in der Regel zum Absturz des entsprechenden Programms, wenn nicht des ganzen Betriebssystems.

Programmieren mit Zeigern ist daher nicht ganz ungefährlich und vielleicht der kritischste Aspekt der C-Programmierung. Nicht unbedingt, weil hier die meisten Fehler passieren, sondern weil diese Fehler die gefährlichsten Auswirkungen haben.

Außerdem gelten Zeiger als ein schwer zu verstehendes Konzept, obwohl sie wirklich nur Variablen sind, deren Werte Speicheradressen sind.

2.1.1 Adress- und Verweisoperator

In dem Beispiel

```
int a = 10;
int b;

b = *(&a);
```

wird mit `&a` die Adresse des Speicherplatzes der Variable `a` ermittelt. Der (unäre!) Verweisoperator `*` (zu unterscheiden von dem binären Multiplikations-Operator `*`) nimmt diese Speicherplatzadresse und gibt den Wert zurück, der dort im Speicher steht, also `10`. Dieser Wert wird dann der Variable `b` zugewiesen.

Diese Beschreibung war nicht ganz korrekt, weil sie eine Frage offenlässt: Woher „weiß“ der Verweis-Operator, dass an dieser Adresse ein `int` steht? (Wenn er das nicht wüsste, würde der den Speicherinhalt mit hoher Wahrscheinlichkeit falsch interpretieren, z.B. als `double`.) Die Anwort ist: Es gibt nicht nur einen Typ von Adressen, sondern viele, z.B. den Typ der „Adresse eines `int`s“. Obwohl *jede* Speicherplatzadresse normalerweise eine einfache, vorzeichenlose 4 Byte große Ganzzahl ist, müssen also verschiedene Typen von Adressen unterschieden werden, z.B. Adressen von `char`s, `short`s, `double`s etc.

Dann sieht die Erklärung des Beispiels so aus: `&a` gibt die Adresse eines `int`s zurück, weil `a` vom Typ `int` ist. Der Verweis-Operator bekommt die Adresse von einem `int` und interpretiert deshalb den Speicherplatzinhalt an dieser Adresse als `int` und gibt diesen `int`-Wert zurück, der dann `b` zugewiesen wird.

2.1.2 Definitionen von Zeigern

Zeiger sind Variablen, die Adressen speichern. Da es verschiedene Typen von Adressen gibt (Adresse eines `int`s, eines `double`s, etc.), muss es auch verschiedene Typen von Zeigern geben. Um etwas Systematik in die Typangabe von Adressen (und damit Zeigern) zu bringen, wird der Verweisoperator in der Typbezeichnung verwendet, nach folgender Regel (von Andrew Koenig): *Variablen werden so definiert, wie sie verwendet werden.* Ein Beispiel:

```
int *c;
```

definiert einen Zeiger auf `int`s. D.h. die Variable `c` speichert Adressen von `int`s. Wird der Verweisoperator auf `c` angewendet, also `*c`, dann nimmt sich der Verweisoperator die Adresse, die in `c` gespeichert ist, und

holt sich an dieser Adresse ein `int` aus dem Speicher, das er zurückgibt. Die Zeile

```
int *c;
```

kann also so gelesen werden: Die Variable `c` wird so definiert, dass wenn der Verweisoperator auf `c` angewendet wird, ein `int` rauskommt. Das ist konsistent mit

```
int a;
```

Denn die letzte Zeile heißt einfach: `a` wird so definiert, dass im Programm für `a` ein Wert vom Typ `int` eingesetzt wird.

C-Datentypen, insbesondere die von Zeigern, können gelegentlich relativ seltsam aussehen, aber es gilt *immer* die Regel: *Variablen werden so definiert, wie sie verwendet werden.*

Das Beispiel von oben kann jetzt so geschrieben werden:

```
int a = 10;
int b;
int *c;

c = &a;
b = *c;
```

Mit `c = &a;` wird die Adresse der Variable `a` in `c` gespeichert. Mit `*c` wird an dieser Adresse ein `int` aus dem Speicher gelesen, das dann in `b` gespeichert wird. Das Beispiel ist also nur eine komplizierte Schreibweise von

```
int a = 10;
int b;

b = a;
```

Noch ein Beispiel:

```
double a = 3.1415;
int b;
double *c;

c = &a;
b = *c;
```

Hier wird `c` als Zeiger vom Typ „Adresse eines `double`s“ definiert. `c = &a;` speichert die Adresse von `a` in `c`. `*c` holt an dieser Adresse ein `double` aus dem Speicher, also `3.1415`, und gibt dieses `double` zurück. `b` ist aber vom Typ `int`, deshalb wird der Wert `3.1415` vor der Zuweisung in ein `int` umgewandelt, d.h. zu `3` abgerundet, und dann in `b` gespeichert.

Und noch ein Beispiel:

```
int a = 10;
int b;
int *c;

c = &b;
*c = a;
```

Hier wird zunächst mit `c = &b;` in `c` die Adresse von `b` gespeichert. Das ist möglich und sinnvoll, obwohl im Speicherplatz von `b` noch kein definierter Wert steht, weil die Adresse des Speicherplatzes von `b` nach der Definition von `b` sehr gut definiert ist. Dann wird mit `*c = ...` der Speicherplatz, dessen Adresse in `c` enthalten ist, überschrieben. In diesem Beispiel wird mit `*c = a;` in diesen Speicherplatz der Wert von `a` gespeichert. In `c` war aber die Adresse von `b`, deshalb ist `*c = a;` äquivalent zu `b = a;`

Was wäre passiert, wenn wir vergessen hätten mit `c = &b;`, die richtige Adresse zu setzen, und eine zufällig Zahl in `c` stehen würde? Dann würde der Wert von `a` in eine zufällig Speicheradresse geschrieben. Das geht oft gut, weil normalerweise nur ein Teil des Speichers benutzt wird, und auch davon nur ein Teil wirklich kritische Daten enthält. Gelegentlich hat es aber auch fatale Folgen, zum Beispiel den Absturz des Programms.

Wieviele Bytes werden benötigt, um eine Adresse eines `int`s zu speichern? Das können wir mit `sizeof(`*Typ der Adresse eines* `ints)` erfahren, aber wie wird der Typ einer Adresse eines `int`s angegeben? Dazu wieder eine einfache Regel: *Die Angabe eines Datentyps ergibt sich aus der Definition einer Variable dieses Typs ohne den Variablennamen und ohne den Strichpunkt.* Da `int *c;` eine Variable vom Typ Adresse eines `int`s definiert, ist `int *` der Typ Adresse eines `int`s. D.h. `sizeof(int *)` ergibt die benötigte Bytegröße, um eine Adresse zu speichern (meistens vier Bytes).

Zur Typkonvertierung wird auch die gleiche Typangabe verwendet. Das heißt eine Speicheradresse eines anderen Typs (z.B. `double *`) kann der Typkonvertierung `(int *)` in eine Adresse vom Typ Adresse eines `int`s konvertiert werden.

2.1.3 Adressen als Rückgabewerte

Ähnlich wie für Variablendefinitionen gibt es auch für Funktionsdefinitionen eine einfache Regel: *Funktionen werden so definiert, wie sie verwendet werden.* D.h. ein Funktionskopf

```
int f(int a)
```

sollte so gelesen werden: Wenn eine Funktion namens `f` mit `f(...)` aufgerufen wird, dann ergibt das ein `int`. Entsprechendes gilt für Adressen als Rückgabewerte. Z.B. kann

```
int *f(int a)
```

so interpretiert werden: Wenn der Verweisoperator mit `*f(...)` (äquivalent zu `*(f(...))`) auf den Rückgabewert der Funktion namens `f` angewendet wird, dann ergibt das ein `int`. D.h. `f()` gibt die Adresse eines `int`s zurück.

2.1.4 Adressen als Funktionsargumente

Eine wichtige Anwendung von Adressen entsteht bei Funktionen. Nehmen wir als Beispiel die Funktion

```
void f(int a)
{
   a = 5;
}
```

Und einen Funktionsaufruf dieser Art:

```
int b = 0;

f(b);
```

Was passiert hier? `b` wird auf den Wert `0` initialisiert, deswegen entspricht `f(b)` dem Aufruf `f(0)`. Beim Funktionsaufruf wird dann eine *neue* Variable `a` definiert, die mit dem Funktionsargument, also `0`, initialisiert wird. Mit `a = 5;` wird dieser Wert überschrieben und eine `5` in `a` gespeichert. Dann wird die Funktion beendet und die lokale Variable `a` deshalb ungültig. Der Programmablauf wird hinter `f(b);` fortgesetzt. Welchen Wert hat dann `b`? Natürlich immer noch `0`, der Funktionsaufruf hat daran gar nichts ändern können und das ist gut so.

Allerdings wäre es manchmal nützlich mit Hilfe einer Funktion irgendwie den Inhalt einer Variable ändern zu können. (Ähnlich wie die In- und Dekrement-Operatoren den Inhalt von Variablen ändern können.) Mit Hilfe von Adressen und Zeigern geht das tatsächlich. Die neue Funktion

```
void g(int *a)
{
   *a = 5;
}
```

bekommt als Argument eine Adresse auf ein `int`. Bei einem Funktionsaufruf wird eine Variable `a` definiert und diese Adresse in `a` gespeichert. (`a` ist also ein Zeiger.) Mit `*a = 5;` wird anschließend der Wert `5` in den Speicherplatz geschrieben, dessen Adresse der Funktion übergeben wurde. D.h. *jetzt* kann mit

```
int b = 0;

g(&b);
```

die Adresse von `b` übergeben werden. In `g()` wird dann ein Zeiger namens `a` definiert und mit dieser Adresse initialisiert. Dann wird in den Speicherplatz von `b` eine `5` geschrieben und die Funktion wieder verlassen. Nach dem Funktionsaufruf `g(&b);` hat sich also der Wert von `b` tatsächlich verändert.

Wenn wie hier an eine Funktion Variablen übergebenen werden und die Funktion die Werte der Variablen ändern kann, wird das auch als „Argumentübergabe per Referenz“ bezeichnet, was hier nichts anderes heißt, als dass die Adresse einer Variable statt ihres Wertes übergeben wird. Dazu noch ein nützliches Beispiel. Die Funktion

```
void swap_ints(int *x, int *y)
{
   int temp;

   temp = *x;
   *x = *y;
   *y = temp;
}
```

bekommt zwei Adressen auf `int`s, holt sich den Wert, der am Speicherplatz mit der ersten Adresse steht, und speichert ihn in der lokalen Variable `temp` zwischen. Dann schreibt sie mit `*x = *y;` den Wert aus dem Speicherplatz an der zweiten Adresse in den vorher ausgelesenen Speicherplatz. Und schließlich wird in den Speicherplatz an dieser zweiten Adresse mit `*y = temp;` der gerettete Wert geschrieben, der früher am Speicherplatz der ersten Adresse stand. Der Effekt ist also, dass die `int`s, deren Adressen übergeben wurden, gerade vertauscht wurden. Mit

```
int a = 2;
int b = 3;

swap_ints(&a, &b);
```

werden also die Werte von `a` und `b` vertauscht, so dass `a` danach den Wert `3` und `b` den Wert `2` bekommt.

2.1.5 Arithmetik mit Adressen

Wie erwähnt werden Adressen intern als (meist vier Byte große) Ganzzahlen verarbeitet. Deswegen ist es möglich eine Adresse auf den Wert `0` zu setzen. Diese `0`-Adresse wird in C meistens als `NULL` geschrieben und dient dazu, deutlich zu machen, dass ein Zeiger *keine* gültige Adresse enthält. (Um `NULL` zu verwenden, sollte zum Beispiel `stddef.h` oder `stdio.h` `#included` werden.)

Um zu überprüfen, ob ein Zeiger eine gültige Adresse oder `NULL` enthält, muss mit `NULL` verglichen werden. Das ist möglich weil `==` und `!=` mit Adressen genauso funktioniert wie mit Ganzzahlen. Auch `<`, `<=`, `>`, `>=` sind möglich.

Es ist sogar möglich, Ganzzahlen zu Adressen zu addieren bzw. den De- oder Inkrementoperator auf Zeiger anzuwenden. Dabei muss aber beachtet werden, dass *nicht* einfach die Ganzzahl zur Adresse addiert wird. Das würde auch nicht viel Sinn machen, denn wenn zur Byteadresse einer `double`-Variable `1` addiert wird, steht an der so erhöhten Adresse ganz sicher kein sinnvolles `double`, da jedes `double` mehr als ein Byte belegt. Die neue Adresse würde also mitten in die Bytes eines `double`s zeigen. Vielmehr wird zur Speicheradresse die Ganzzahl multipliziert mit der Bytegröße des entsprechenden Datentyps addiert (also im Beispiel oben `sizeof(double)`). Ebenso erhöht `++` eine Zeigervariable nicht um `1` sondern um die Bytegröße des Datentyps, auf den der Zeiger zeigt. Analog erniedrigt `--` um diese Größe.

Ein Beispiel zu einfacher Adressarithmetik:

```
double *a = NULL;

a = malloc(10 * sizeof(double));

if (NULL != a)
{
   *a = 0.0;
   *(a + 1) = 3.1415;
   *(a + 2) = *(a + 1);

   free(a);
}
```

Die Funktionen `malloc()` und `free()` werden zum Beispiel in `stdlib.h` oder `malloc.h` deklariert, und dienen dazu Speicher „dynamisch“ zu reservieren, bzw. freizugeben. `malloc()` erwartet als Argument eine Ganzzahl, die angibt, wieviele Bytes reserviert werden sollen. `malloc()` gibt dann entweder `NULL` zurück, falls der Speicher nicht reseviert werden konnte, oder die Adresse des reservierten Speicherblocks. Der Typ der Rückgabeadresse ist `void *`, also die Adresse eines unbekannten Datentyps. (`malloc()` kann nicht wissen, welche Daten in den Speicherblock geschrieben werden sollen.) Adressen vom Typ `void *` verhalten sich ähnlich wie andere Adressen, aber es kann keine Adressarithmetik durchgeführt werden, weil `void` keine Bytegröße hat. Die `void *`-Adresse wird bei der Zuweisung von `a` aber ohne weiteres in eine `double *`-Adresse umgewandelt.

Das Gegenstück zu `malloc()` ist `free()` und dient dazu den Speicherblock an einer angegebenen Adresse wieder freizugeben. Das ist wichtig, damit immer genug Speicher für neue `malloc()`s zur Verfügung steht.

Im Beispiel wird ein Speicherblock für `10` `doubles` angefordert. Die benötigte Bytegröße wird dabei mit dem `sizeof`-Operator berechnet. (Übrigens wird praktisch jeder C-Compiler die Multiplikation `10 * sizeof(double)` schon während des Compilierens ausführen und nur noch das Ergebnis in den Programmcode einsetzen, da der Compiler `sizeof(double)` kennt und das Produkt zweier Konstanten immer noch eine Konstante ist, also nicht immer neu berechnet werden muss.)

Nach dem Reservieren des Speichers und der Speicherung der zurückgegebenen Adresse in `a` wird abgefragt, ob `a` ungleich `NULL` ist, d.h. ob tatsächlich Speicher reserviert wurde. Wenn dem so ist, wird an den Anfang des Speicherblocks mit `*a = 0.0` ein `double` vom Wert `0.0` gespeichert. Aber in dem Speicherblock ist noch Platz für 9 weitere `double`s. Mit `a + 1` wird deshalb die Adresse hinter dem ersten `double` berechnet. (Falls ein `double` 8 Bytes groß ist, ist `a + 1` um 8 größer als die Startadresse des Speicherblocks.) Mit `*(a + 1) = 3.1415;` wird dann an dieser Adresse der Wert `3.1415` gespeichert. In der nächsten Zeile wird dieser Wert mit `*(a + 1)` ausgelesen und gleich mit `*(a + 2) =` ... in die nachfolgenden Bytes im Speicherblock gespeichert. Dann wird der Speicherblock wieder freigegeben. (Ja, das ist ein rein akademisches Beispiel.)

Da die Schreibweise `*(`*Zeiger* `+` *Ganzzahl*`)` etwas umständlich ist, kann sie auch mit `Zeiger[`*Ganzzahl*`]` abgekürzt werden. Das Beispiel würde dann so aussehen:

```
double *a = NULL;
```

```
a = malloc(10 * sizeof(double));

if (NULL != a)
{
   a[0] = 0.0;
   a[1] = 3.1415;
   a[2] = a[1];

   free(a);
}
```

Das kann folgendermaßen interpretiert werden: `a` ist der Name einer Reihe von `double`s, die hintereinander im Speicher stehen. Mit `a[`*Index*`]` wird auf ein Element in dieser Reihe zugriffen, bzw. mit `a[`*Index*`] =` ... der Wert eines Elements gesetzt. Deswegen wird `[]` auch als Arrayzugriff interpretiert. Aber das ist nur eine Interpretation, tatsächlich bedeutet `a[`*Index*`]` nichts anderes als `*(a +` *Index*`)`. (Und das ist das Gleiche wie `*(`*Index*`+ a)` und deswegen auch das Gleiche wie *Index*`[a]`!)

2.1.6 Zeiger auf Zeiger

Zeiger sind Variablen, haben also auch einen Speicherplatz in dem sie ihren Wert (eine Adresse) speichern. Dieser Speicherplatz hat auch eine Adresse und zwar vom Typ Adresse auf eine Adresse auf einen bestimmten Datentyp. Beispiel:

```
int a = 42;
int *b;

b = &a;
```

`b` speichert hier die Adresse eines `int`s. Deswegen ist die Adresse der Variable `b`, also `&b` vom Typ Adresse einer Adresse eines `int`s. Wenn wir diese Variable in einer Variable `c` speichern wollen, müssen wir wissen wie dieser Typ in C formuliert wird. Dazu benutzen wir einfach die Regel von Koenig: *Variablen werden so definiert, wie sie verwendet werden.* Wir wissen, dass `c` eine Adresse einer Adresse eines `int`s speichern soll, d.h. der Wert `*c` ist die Adresse eines `int`s. Dann ist aber `*(*c)` (oder gleichbedeutend `**c`) ein `int`-Wert. Wenn aber `**c` ein `int` ist, dann ist `int **c;` die richtige Definition. Also können wir mit

```
int *b;
int **c;

c = &b;
```

in `c` die Adresse des Speicherplatzes von `b` speichern. (In dem Speicherplatz von `b` muss dazu kein definierter Wert stehen.)

Der Typ einer Adresse einer Adresse eines `int`s ergibt sich wieder durch Weglassen des `c;` in `int **c;`, ist also `int **`. Analog wird der Typ von Adressen von Adressen von Adressen von `int`s mit drei `*` angegeben. Da so viele Indirektionen das Vorstellungsvermögen doch erheblich beanspruchen, sind solche Typen aber eher selten.

2.1.7 Adressen von Funktionen

Der C-Compiler übersetzt Funktionsdefinitionen in ausführbare Maschinenbefehle. Die müssen aber auch im Speicher abgelegt werden, damit der Prozessor sie ausführen kann. Daher hat zur Laufzeit jede Funktionsdefinition eine definierte Adresse. Erhalten kann ein Programm diese Adresse mit dem Adressoperator: &*Funktionsname* ergibt also die Adresse der Funktion namens *Funktionsname*.

Wozu sind Adressen von Funktionen gut? Zum Beispiel um die Funktion, deren Maschinenbefehle an der angegebenen Adresse im Speicher stehen, aufzurufen. Das funktioniert mit dem unären Verweisoperator `*`. Allerdings müssen bei *jedem* Funktionsaufruf Argumente übergeben werden. Dazu wird der Funktionsaufruf-Operator `()` benutzt. Dabei muss beachtet werden, dass der Funktionsaufruf-Operator Vorrang vor dem Verweisoperator hat („weil“ binäre Operatoren, die so wie der Funktionsaufruf ohne Leerzeichen geschrieben werden, Vorrang vor unären Operatoren wie dem Verweisoperator haben). Deswegen müssen wir Klammern setzen, um zuerst den Verweisoperator auf die Funktionsadresse wirken zu lassen. Nehmen wir wieder unsere `verdopple()` Funktion:

```
int verdopple(int a)
{
   return(2 * a);
}
```

Jetzt kann die Funktion mit

```
int b;

b = (*(&verdopple))(100);
```

aufgerufen werden. Zur Wiederholung: `&verdopple` ist die Adresse der Funktion, `(*(&verdopple))` ergibt „die Funktion selbst“, und `(*(&verdopple))(100)` ist dann der komplette Funktionsaufruf, der äquivalent zu `verdopple(100)` ist.

Wie können wir einen Zeiger `c` auf eine Funktion definieren? Entsprechend der Regel „*Variablen werden so definiert, wie sie verwendet werden*“ gehen wir so vor: `c` enthält die Adresse einer Funktion, die ein `int` als Argument bekommt und ein `int` zurückgibt. Dann ist `*c` „die Funktion selbst“, und mit dem Funktionsaufruf-Operator heißt der Funktionsaufruf dann `(*c)(int)`. (An dieser Stelle müssen in der Argumentliste keine Variablennamen aufgeführt werden, aber die Typen der Variablen.) Dieser Funktionsaufruf ergibt aber ein `int`. Deswegen ist `int (*c)(int);` die richtige Definition für `c` und wir können schreiben:

```
int b;
int (*c)(int);

c = &verdopple;
b = (*c)(100);
```

Der Typ einer Funktionsadresse ergibt sich dann wieder durch Weglassen des Variablennamens und des Strichpunkts. Als Typ „Adresse einer Funktion, die ein `int` als Argument erwartet und ein `int` zurückgibt,“ ergibt sich dann: `int (*)(int)`. Das sieht auf den ersten Blick etwas albern aus, aber weil wir uns strikt an die Regeln gehalten haben ist das richtig und `sizeof(int (*)(int))` gibt dann auch brav die übliche Bytegröße von Adressen zurück. Typkonvertierungen in diesen Typ funktionieren entsprechend mit `(int (*)(int))`.

Bei Funktionsadressen sind die Regeln, wie Variablen definiert werden, und sich daraus Typen ergeben, unentbehrlich. Damit können dann auch Ungetüme, wie Zeiger auf Funktionen, die Adressen von Funktionen zurückgeben, definiert werden.

Etwas verwirrend bei Funktionsadressen ist Folgendes: Die Adress- und Verweisoperatoren sind nicht immer notwendig, denn wenn ein Funktionsname ohne Funktionsaufruf-Operator, also ohne nachfolgende Klammern, erscheint, kann nur die Adresse der Funktion gemeint sein. Also kann statt `c = &verdopple;` auch `c = verdopple;` geschrieben werden. Ebenso: Wenn eine Adresse auf eine Funktion *mit* nachfolgenden Klammer erscheint, kann nur ein Funktionsaufruf der Funktion selbst gemeint sein. Also kann statt `(*c)(100)` auch `(c)(100)` oder `c(100)` geschrieben werden. Das gilt aber *nicht* für die Definition von Funktionszeigern und den Typ von Funktionsadressen. Deswegen sollte darauf verzichtet werden, diese syntaktischen Möglichkeiten auszunutzen.

2.2 Arrays

Ähnlich wie lokale Variablen definiert werden können, also Speicher für einen Wert eines bestimmen Datentyps reserviert werden kann, kann auch Speicher für mehrere Werte eines bestimmten Datentyps reserviert werden. Die Werte werden dabei direkt hintereinander im Speicher abgelegt. Diese Konstruktion wird Array genannt und unterliegt den gleichen Gültigkeits- und Sichtbarkeitsregeln wie alle lokalen Variablen.

2.2.1 Eindimensionale Arrays

Ein Array wird (wie alle Variablen) so definiert, wie es verwendet wird. Wenn also `a` ein Array von `int`s ist, dann ist z.B. `a[0]` das 0. Element dieses Arrays (Vorsicht: Zählen in C beginnt mit 0!). Da jedes Element des Arrays ein `int` ist, ergibt sich die Definition `int a[0];`. Die `0` als Index macht hier nicht soviel Sinn, deswegen wird an dieser Position die Anzahl der Element des Arrays angegeben. `int a[6];` würde also ein Array definieren, das Platz für 6 `int`s bietet. Da das Zählen in C mit 0 beginnt, werden die Elemente mit `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]` und `a[5]` angesprochen. Mehr Speicherplatz ist nicht reserviert, d.h. `a[6]` würde über den reservierten Speicherplatz hinauszeigen. Das Problem in C ist nun, dass bei solchen sogenannten index-out-of-bounds-Fehlern keine Fehlermeldungen erzeugt werden, genauso wie Zeiger auf willkürliche Adressen nicht direkt Fehler erzeugen.

Ein anderer Nachteil in C ist, dass die Größe des Arrays dem Compiler bekannt sein muss, d.h. es darf keine Variable als Größe in einer Arraydefinition verwendet werden. Hier ein Beispiel:

```
double a[5];

a[0] = 3.1415;
a[1] = a[0];
```

Dabei wird Speicher für 5 `double`s reserviert und in die 0. und 1. Position der Wert `3.1415` geschrieben.

Entsprechend der Größe des für das Array reservierten Speicherblocks ergibt `sizeof(a)` in diesem Fall 5 mal die Bytegröße eines `double`s. Die Größe eines Arrays zu bestimmen ist eine von zwei Sachen, die mit Arrays gemacht werden können. Die andere Sache ist, die Adresse des 0. Elements zu bestimmen.

Die Adresse des 0. Elements des Arrays `a` kann als `&a[0]` geschrieben werden. (Das ist äquivalent zu `&(a[0])` weil der binäre Arrayzugriff-Operator (ohne Leerzeichen!) Vorrang vor dem unären Adress-Operator hat.) Aber es ist genauso richtig, einfach `a` zu benutzen, denn der Arrayname steht *immer* für die Adresse des 0. Elements, *außer* er steht in `sizeof`.

Wie bei der Adressarithmetik erwähnt, ist `a[1]` äquivalent zu `*(a + 1)`. Der Arrayname `a` steht dabei für die Adresse des 0. Elements (die Adresse eines `int`s), zu der mit `a + 1` entsprechend der Adressarithmetik die Bytegröße eines `int`s addiert wird. `*(a + 1)` ergibt dann den Speicherinhalt an dieser Adresse, die auch als Adresse des 1. Elements des Arrays `a` angesehen werden könnte.

In diesem Sinne gibt es also gar keinen „Arrayzugriff“, sondern nur die Abkürzung *Arrayname*`[`*Index*`]` für `*(`*Adresse-des-0.-Elements-des-Arrays* `+` *Index*`)`. Daraus ergibt sich auch, warum C nicht überprüft, ob ein Index größer als erlaubt ist. Denn Adressarithmetik kennt solche Überprüfungen nicht.

2.2.2 Initialisierung von Arrays

Arrays können bequem initialisiert werden. Dazu werden die Elementwerte in geschweiften Klammern als Liste angegeben. Zum Beispiel:

```
int a[5] = {2, 3, 5, 7, 11};
```

Diese Definition reserviert nicht nur Speicher für fünf `int`s, sondern schreibt auch die fünf angegebenen Werte in den reservierten Speicher. Da der C-Compiler die Werte auch abzählen kann, muss hier in der Definition, die Größe des Arrays nicht explizit angegeben werden, d.h. es ist auch möglich

```
int a[] = {2, 3, 5, 7, 11};
```

zu verwenden. Arrays von `char`s werden auch Zeichenketten oder Strings genannt und können ähnlich initialisiert werden:

```
char a[] = {'H', 'a', 'l',
   'l', 'o', '!'};
```

Viele Bibliotheksfunktionen erwarten aber Zeichenketten, die mit dem `'\0'`-Zeichen abgeschlossen werden. (Das `'\0'`-Zeichen hat praktisch *immer* den ASCII-Code `0`.) Deswegen ist es besser ein `'\0'` anzuhängen:

```
char a[] = {'H', 'a', 'l',
   'l', 'o', '!', '\0'};
```

Dafür gibt es auch eine Abkürzung:

```
char a[] = "Hallo!";
```

Das ist aber etwas ganz anderes als:

```
char *a = "Hallo!";
```

Im letzten Beispiel wird kein Array definiert, sondern ein Zeiger auf `char`s, d.h. es wird nur der Speicherplatz für eine Adresse reserviert und darin die Adresse einer Zeichenkette gespeichert, die der Compiler irgendwo in das Programm geschrieben hat. Diese Adresse bleibt bei allen Ausführungen dieser Definition die gleiche, während bei der Definition eines Arrays immer wieder ein anderer Speicherplatz reserviert werden kann. Das ist ganz ähnlich wie in

```
printf("Hallo!\n");
```

Hier wird auch die Zeichenkette `"Hallo!\n"` vom Compiler irgendwo im Programmcode abgelegt und die Adresse dann an `printf()` übergeben. Entsprechend kann auch mit

```
printf(a);
```

das oben definierte `char`-Array ausgegeben werden, indem seine Adresse an `printf()` übergeben wird. Hier ist ganz entscheidend, dass ein `'\0'`-Zeichen das Ende des Strings markiert, weil `printf()` sonst munter alles ausgibt, was dahinter im Speicher steht, bis es zufällig auf ein `'\0'`-Zeichen trifft. Das ist allerdings nicht ganz sauber, weil `printf()` sein erstes Argument als *Formatstring* interpretiert und deswegen nicht einfach die Zeichen der Reihe nach ausgibt. Sicherer ist es deshalb mit dem Formatelement `%s` für Strings zu schreiben:

```
printf("%s", a);
```

2.2.3 Mehrdimensionale Arrays

Wie wir gesehen haben, gibt es eigentlich keine Arrays in C, sondern nur Adressen und Speicherblöcke und eine abkürzende Schreibweise für die Adressarithmetik.

Ähnliches gilt auch für mehrdimensionale Arrays in C: Es gibt sie eigentlich nicht. Der Ersatzmechanismus soll an einem Beispiel erklärt werden:

Ein zweidimensionales Feld von Zahlen, z.B. eine 3×4 Matrix, kann auch in einem eindimensionalen Array der Größe 12 gespeichert werden. Nach der Definition

```
double m[3 * 4];
```

kann mit $m[i * 4 + j]$ der Eintrag in der `i`-ten Zeile und der `j`-ten Spalte angesprochen werden. (Wieder wird bei `0` angefangen zu zählen, also geht `i` von 0 bis 2 und `j` von 0 bis 3.) Natürlich sind „Zeile“ und „Spalte“ keine gut definierten Begriffe hier. Wichtiger ist die Unterscheidung zwischen „äußerem“ (hier: `i`) und „innerem“ (hier: `j`) Index. Positionen, die sich im inneren Index um 1 unterscheiden, sind auch im Speicher benachbart. Unterscheiden sich die Positionen in einem äußeren Index um 1, sind sie im Speicher deutlich weiter entfernt. (In diesem Beispiel um 4 Speicherpositionen für `double`s, wie an dem Ausdruck `i * 4 + j` zu sehen ist.) (Vorsicht: Andere Autoren verwenden die Begriffe *innere* und *äußere Indizes* gerade umgekehrt.)

Die abkürzende Schreibweise für dieses Beispiel ist:

```
double m[3][4];
```

Dann kann mit `m[i][j]` auf das Element, das an der `i * 4 + j`-ten Speicherposition für `double`s steht, zugegriffen werden. Die nötige Multiplikation und Addition wird also genauso durchgeführt wie vorher. Um es genau zu sagen: `m[i][j]` ist die Abkürzung für `*(&m[0][0] + i * 4 + j)`, wobei `&m[0][0]` die Adresse des `double`s an der 0. Position im Speicherblock ist.

Unterschiede zwischen dieser Schreibweise und dem explizit eindimensionalen Array ergeben sich durch den Typ und dadurch auch für den `sizeof`-Operator. Was ist der Typ von `m` nach einer Definition als `double m[3][4];`? Nach unserer Regel (Variablenname und Strichpunkt weglassen) ist das `double [3][4]`. Für `sizeof()` muss auch wirklich dieser Typ übergeben werden. Wird ein Array als Funktionargument übergeben, dann wird aber immer nur die Adresse des 0. Elements übergeben, deshalb spielt die Dimension des äußersten Index keine Rolle. (Im Beispiel taucht die `3` in der Berechnung `i * 4 + j` nicht auf.) Daher kann die Dimension des äußersten Index für eine Typangabe in einem Funktionskopf weggelassen werden.

Oben wurde die Adresse des ersten `double`s als `&m[0][0]` (äquivalent zu `m[0]`) statt `m` geschrieben, weil `m` gleich `&m[0]` ist. `m[0]` ist aber ein Array aus 4 `double`s. Der Typ von `m[0]` ist deshalb `double [4]` und die Größe von diesem Datentyp ist 4 mal `sizeof(double)`. Dieser Größe würde auch bei der Adressarithmetik mit `&m[0]` verwendet werden, was nicht gut ist, wenn alle einzelnen `double`s angesprochen werden sollen.

Dagegen ist der Ausdruck `m[0]` die Adresse seines 0. Elements, also `&m[0][0]` und das ist die Adresse eines `double`s. Also wird bei der Adressarithmetik mit `m[0]` oder `&m[0][0]` die Bytegröße eines `double`s verwendet.

Höherdimensionale Arrays funktionieren ganz analog, z.B. ist nach

```
double m[2][3][4];
```

der Ausdruck `m[i][j][k]` äquivalent zu `*(&m[0][0][0] + i * 3 * 4 + j * 4 + k)`.

2.2.4 Arrays von Zeigern

Flexibler als mehrdimensionale Arrays sind Arrays von Zeigern. Z.B.:

```
double *m[3];

m[0] = (double *)malloc(4 *
   sizeof(double));
m[1] = (double *)malloc(4 *
   sizeof(double));
m[2] = (double *)malloc(4 *
   sizeof(double));
```

Hier wird ein Array aus drei Zeigern auf `double`s definiert. Jedem Zeiger wird die Adresse eines mit `malloc()` reservierten Speicherblocks zugewiesen, der jeweils 4 `double`s aufnimmt. `m[i]` ist die Adresse eines solchen Speicherblocks, deswegen kann mit `m[i][j]` bzw. `*(m[i] + j)` ein `double` angesprochen werden. Der Vorteil der Verwendung von `malloc()` ist, dass die Speicherblöcke `m[0]`, `m[1]` und `m[2]` auch unterschiedlich groß sein dürfen und die Größe beim Programmieren noch nicht feststehen muss. Um auch die erste Dimension (also die 3) frei angeben zu können, muss auch das Array von Zeigern mit `malloc()` reserviert werden:

```
double **m;

m = (double **)malloc(3 *
```

```
   sizeof(double *));
m[0] = (double *)malloc(4 *
   sizeof(double));
m[1] = (double *)malloc(4 *
   sizeof(double));
m[2] = (double *)malloc(4 *
   sizeof(double));
```

Hier ist `m` also ein Zeiger auf Adressen auf `double`s. Das erste `malloc()` reserviert Speicher für 3 Adressen auf `double`s (deswegen `sizeof(double *)`) und die restlichen `malloc()`s reservieren wieder die Speicherblöcke für je 4 `double`s. Die Adressarithmetik macht es möglich, dass der Zugriff auf einzelne Elemente noch mit `m[i][j]` möglich ist.

Diese Art ein Feld von Zahlen zu verwalten erlaubt die größte Flexibilität, aber erfordert auch einiges an Programmieraufwand zum Reservieren und späterem Freigeben von Speicher. In den Beispielen fehlen außerdem Abfragen, ob `malloc()` überhaupt erfolgreich war und nicht etwa eine `NULL`-Adresse zurückgegeben hat.

2.2.5 Übergabe von Kommandozeilenargumenten

Eine weitere Anwendung von Zeigerarrays ist die Übergabe der Kommandozeilenargumente des Aufrufs eines compilierten Programms. Diese Argumente können der Funktion `main()` übergeben werden, dazu ein Beispiel, das die Argumente ausgibt:

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int i;

   for (i = 0; i < argc; i++)
   {
      printf("%s ", argv[i]);
   }
   printf("\n");

   return(0);
}
```

`main()` bekommt dabei immer ein `int`, das die Anzahl der Argumente enthält (plus 1, weil das 0. der Programmname ist), und eine Adresse eines Arrays mit Zeigern auf `char`s. Letzters kann als `char *argv[]` oder äquivalent `char **argv` angegeben werden, da Arrays immer nur als Adressen ihres 0. Elements übergeben werden.

In dem Beispiel werden alle Elemente `argv[i]`, also Adressen eine `char`s an `printf()` übergeben. Diese Adressen geben nicht nur einzelne `char`s an, sondern Arrays von Zeichen, die mit dem Zeichen `'\0'` beendet sind. Das Formatelement für solche `'\0'`-terminierten Zeichenketten ist `%s` und steht im Formatstring, der als erstes Argument an `printf()` übergeben wird.

2.3 Datenstrukturen

2.3.1 struct-Deklarationen und -Definitionen

Arrays sind dafür geeignet, Felder von Werten gleichen Datentyps zusammenzufassen. Falls Werte unterschiedlichen Datentyps zusammengefasst werden sollen, werden in C sogenannte `struct`s eingesetzt. Ein `struct`-Datentyp muss zunächst deklariert werden. Dabei wird noch kein Speicher reserviert, sondern nur der `struct`-Datentyp spezifiziert. (Sobald Speicher belegt oder reserviert wird, wird ein Vorgang Definition statt Deklaration genannt.) Mit

```
struct Kreis
{
   double m_x;
   double m_y;
   double r;
};
```

wird zum Beispiel ein `struct`-Datentyp namens `Kreis` deklariert. Jede Variable dieses Datentyps enthält die x- und y-Koordinate des Mittelpunkts `m_x` und `m_y`, sowie den Radius `r`.

Nach dieser Deklaration kann mit

```
struct Kreis k;
```

eine Variable vom Typ `struct Kreis` definiert werden. Dabei wird für `k` wieder ein Stück Speicher reserviert, dessen Größe mit `sizeof(struct Kreis)` abgefragt werden kann.

2.3.2 Zugriff auf Elemente von structs

Zugriff auf die Elemente der Struktur ist dann über den binären `.`-Operator möglich:

```
k.m_x = 0.0;
k.m_y = k.m_x;
k.r = 10.0;
```

Die Struktur-Deklaration und Definition einer Variable kann auch zusammengefasst werden:

```
struct Kreis
{
   double m_x;
   double m_y;
   double r;
} k;
```

2.3.3 Zeiger auf structs

Bei der Übergabe von `struct`-Werten als Argumente von Funktionen und dem Zuweisen von `struct`-Werten werden jeweils alle Elemente kopiert, was etwas langsam ist. Deswegen werden meistens nur Adressen von `struct`-Variablen übergeben, was wesentlich schneller ist. Die Adresse der `struct`-Variable `k` kann mit `&k` ermittelt werden. Wie wird ein Zeiger `k_zeiger` auf eine `struct`-Variable definiert? Da `*k_zeiger` vom Typ `struct Kreis` ist, ist die Definition der Variable `k_zeiger` gegeben durch

```
struct Kreis *k_zeiger;
```

und mit

```
k_zeiger = &k;
```

kann die Adresse von `k` in `k_zeiger` gespeichert werden. Damit lautet das Beispiel von oben:

```
(*k_zeiger).mx = 0.0;
(*k_zeiger).my = (*k_zeiger).mx;
(*k_zeiger).r = 10.0;
```

(Die Klammern sind nötig, da der binäre Operator `.` (keine Leerzeichen!) Vorrang vor dem unären Verweis-Operator hat.) Als Abkürzung kann der `->`-Operator verwendet werden. Damit würden die Elemente so gesetzt werden:

```
k_zeiger->mx = 0.0;
k_zeiger->my = k_zeiger->mx;
k_zeiger->r = 10.0;
```

2.3.4 Initialisierung von structs

`struct`-Variablen können aber auch bei ihrer Definition initialisiert werden:

```
struct Kreis k = {0.0, 0.0, 10.0};
```

wobei die Reihenfolge der Element in der Deklaration implizit verwendet wird. Bei umfangreicheren `struct`s ist die Zuordnung der Werte zu den Elementen daher nicht ganz einfach, deshalb wird meistens auf eine derartige Initialisierung verzichtet.

2.3.5 Rekursive structs

Noch eine Bemerkung zur `struct`-Deklaration: Wirklich rekursive `struct`s, also `struct`s die sich selbst als Element beinhalten, machen wenig Sinn, weil sie unendlich groß wären. Aber Elemente eines `struct`-Datentyps dürfen auch Zeiger auf die gleiche `struct` sein, was sich insbesondere für vernetzte Datenstrukturen, wie verbundene Listen, Bäume und Graphen, eignet.

2.4 Sonstige Datentypen

In diesem Abschnitt sollen weitere Datentypen erklärt werden. Allerdings ist nur die Möglichkeit eigene Datentypen mit `typedef` zu deklarieren wirklich interessant. Die Konzepte von Bitfeldern, `enum` und `union` werden nur der Vollständigkeit halber erwähnt.

2.4.1 typedef

Inzwischen sollte klar geworden sein, dass Typangaben in C gelegentlich relativ aufwendig werden, z.B. die Datentypen von Funktionsadressen. Es gibt daher die Möglichkeit, für solche komplexen Datentypen einen neuen Namen zu deklarieren. Das wird auch „Typdefinition“ genannt, was kein guter Name ist, weil dabei kein Speicher reserviert wird. Deswegen sollte besser von einer Typdeklaration gesprochen werden.

Hier ein Kochrezept für Typdeklarationen:

1. Man nehme die Definition einer Variable von dem Datentyp, für den ein neuer Name deklariert werden soll, und ersetze den Variablennamen durch den neuen Typnamen. (Für den Typnamen gelten die gleichen Regeln wie für Variablennamen.)

2. Die geänderte Variablendefinition wird nun an das Keyword `typdef` angehängt, womit die Typdeklaration fertig ist.

Damit sind Typdeklarationen nicht schwerer als Variablendefinitionen. (Und für die gilt: *Variablen werden so definiert, wie sie verwendet werden.*)

Als Beispiel soll ein Datentyp namens `fTyp` deklariert werden, der die Adresse einer Funktion beschreibt, die die Adresse eines `char`s als Argument erwartet und die Adresse eines `double`s zurückgibt. Wie würde eine Variable namens `f` dieses Typs *verwendet* werden? Ein Funktionsaufruf funktioniert mit `(*f)(...)`. Als Argument müssen wir hier den Typ Adresse eines `char` einsetzen, also `char *`. Der Funktionsaufruf gibt eine Adresse eines `double` zurück, also `double *`. Deshalb ergibt sich als Variablendefinition:

```
double *(*f)(char *);
```

(Dies kann auch so gelesen werden: Wenn `(*f)(...)` die Adresse eines `double` ergibt, dann ist `*(*f)(...)` ein `double`. Damit kommen wir zur gleichen Definition wie oben.)

In dieser Variablendefinition sollen wir nun den Variablennamen `f` durch den Typnamen `fTyp` ersetzen und `typedef` voranstellen. Also:

```
typedef double *(*fTyp)(char *);
```

Nach so einer Typdeklaration können nun entsprechende Variablendefinitionen mit diesem Typ viel einfacher vorgenommen werden, z.B. definiert

```
fTyp f;
```

einen Funktionszeiger `f` auf eine Funktion, wie sie oben beschrieben wurde.

Noch ein viel einfacheres Beispiel: Der Typ `signed char` wird oft `byte` genannt. Wie sieht eine entsprechende Typdeklaration aus? Wir beginnen einfach mit der Definition einer Variable `a` vom Typ `signed char`:

```
signed char a;
```

Ersetzen des Variablennamens `a` durch `byte` und Voranstellen von `typedef` ergibt die Typdeklaration:

```
typedef signed char byte;
```

womit ein neuer Datentyp `byte` eingeführt wurde, dessen Werte von `-128` bis `127` gehen.

Oft wird eine `struct`- mit einer Typdeklaration zusammengefasst, um gleich den Typ einer Adresse auf die `struct` zu deklarieren, z.B.:

```
typedef struct s
{
   int i;
   double d;
} *s_adresse;
```

Damit wird nicht nur `struct s` deklariert, sondern auch der Typ `s_adresse` als `struct s *`.

2.4.2 Bitfelder

Um Speicherplatz zu sparen, können innerhalb von `struct`s Bitfelder definiert werden, d.h. Variablen die eine individuelle Anzahl von Bits benötigen. Zum Beispiel:

```
struct ampel
{
   int rot_an : 1;
   int gelb_an : 1;
   int gruen_an : 1;
};

struct ampel a;
```

Die Elemente `a.rot_an`, `a.gelb_an` und `a.gruen_an` belegen nur ein Bit und können deshalb nur Werte `0` und `1` annehmen. Bitfelder werden relativ selten eingesetzt; stattdessen wird oft eine Ganzzahl benutzt und die einzelnen Bits mit den bitweisen Operatoren angesprochen.

2.4.3 union

Zu Zeiten, als Speicherplatz noch knapp und teuer war, war es üblich den gleichen Speicherplatz in verschiedenen Programmteilen für verschiedene Daten zu benutzen. Dabei ergibt sich das Problem, dass sich auch der Typ ändert. Ein `union` erlaubt es, verschiedene Datentypen in dem Speicherplatz einer Variable zu speichern. (Wobei zu einem Zeitpunkt immer nur ein Wert in dem Speicherplatz stehen darf.) Die Auswahl des Typs funktioniert dabei syntaktisch so wie bei `struct`s. Zum Beispiel:

```
union Koordinate
{
   long ganzzahlig;
   float fliesskomma;
};

union Koordinate x;
```

Hier wird zunächst eine `union Koordinate` deklariert und dann eine Variable `x` von diesem Typ definiert. Mit `x.ganzzahlig = 123;` kann dann ein `long` in dem Speicherplatz abgespeichert werden und mit `x.fliesskomma = 456.789;` ein `float`. Entsprechend können die Werte auch wieder ausgelesen werden. Adressen, Zeiger und `->` funktionieren wie bei `struct`s.

Nachdem Speicherplatz inzwischen selten ein Problem ist, und die Benutzung von `union` mit relativ großem Aufwand verbunden ist, wird es praktisch nicht mehr verwendet.

2.4.4 enum

Aufzählungen mit `enum` werden sehr selten verwendet, vor allem weil es eine andere, mächtigere Möglichkeit gibt, Konstanten Namen zu geben. (Diese `#define`s werden aber in diesem Kurs erst zusammen mit dem Präprozessor besprochen.) Mit

```
enum Wochentag {Montag, Dienstag,
   Mittwoch, Donnerstag, Freitag,
   Samstag, Sonntag};
```

kann ein Datentyp `enum Wochentag` definiert werden, der nur die Werte `Montag`, ..., `Sonntag` annehmen kann. (Intern werden diese Werte durch Ganzzahlen repräsentiert.) D.h. mit

```
enum Wochentag t = Montag;
```

wird eine Variable vom Typ `enum Wochentag` definiert und mit dem Wert `Montag` initialisiert.

Kapitel 3

Einführung in OpenGL

(Dieses Kapitel ist mehr oder weniger eine Übersetzung des ersten Kapitels des „OpenGL Programming Guide, Third Edition“ von Woo, Neider, Davis und Shreiner.)

3.1 Was ist OpenGL?

Leider gibt es in C und seinen Standardbibliotheken keine Funktionen für graphische Ausgaben. OpenGL dagegen steht für “open graphics library” und bietet eine Vielzahl von Funktionen, um Graphiken zu erstellen.

Das eigentliche OpenGL ist eine sehr hardware-nahe Software-Schnittstelle aus etwa 200 Kommandos und reicht (im Idealfall) diese Kommandos direkt an die Graphik-Hardware weiter. Falls die Graphik-Hardware keine oder nicht alle OpenGL-Kommandos versteht, müssen die fehlenden Kommandos in Software ausgewertet werden, was sehr aufwendig sein kann ist.

Der Vorteil der OpenGL-Schnittstelle ist aber, dass sich der Programmierer (solange Geschwindigkeit keine Rolle spielt) keine Gedanken darüber machen muss, wie die OpenGL-Kommandos tatsächlich ausgewertet werden: Wichtig für den Programmierer sind nur die C-Funktionen, um OpenGL-Kommandos aufzurufen.

In diesem Kurs wird OpenGL nur ganz oberflächlich erklärt, damit überhaupt *zweidimensionale* graphische Beispiele und Übungen möglich sind. Außerdem soll am Beispiel von OpenGL gezeigt werden, wie große Bibliotheken, die nicht zum C Standard gehören, verwendet werden.

Eine tiefergehende Einführung in die Grundlagen der zwei- und dreidimensionalen Computergraphik wird zum Beispiel in der „Grundlagenvorlesung Interaktive Systeme“ von Prof. Thomas Ertl jeweils im Wintersemester angeboten. Spezielle Algorithmen unter Ausnutzung spezifischer OpenGL-Fähigkeiten werden im Sommersemester auch von Prof. Ertl in der „Bildsynthese“–Vorlesung besprochen.

3.2 Aus was besteht OpenGL?

OpenGL besteht aus dem eigentlich OpenGL-Kern und einer Utilities-Bibliothek GLU, die weitere Funktionen enthält, die nicht von der Grafik-Hardware ausgeführt werden können. Darüberhinaus wird in diesem Kurs die Bibliothek GLUT verwendet, die mit den meisten OpenGL-Implementierung mitgeliefert wird, und zum Öffnen von Fenstern sowie zum Abfragen von Benutzeraktionen dient und auch die restliche Kommunikation mit dem Betriebssystem erledigt.

Ein C-Programmierer sieht von OpenGL vor allem die `.h`-Dateien, nämlich `gl.h`, `glu.h` und `glut.h`. Es reicht aber aus `glut.h` zu `#include`n, da diese Datei, `#include`s für die beiden anderen enthält.

Ausserdem müssen die eigentlichen Bibliotheken für OpenGL, GLU und GLUT vorhanden sein und dem C-Compiler bekannt gemacht werden, genauso wie auch die Mathematikbibliothek mit `-lm` angegeben werden musste. Bei den OpenGL-Bibliotheken unter UNIX-Systemen sind üblicherweise folgende C-Compiler-Kommandoschalter notwendig: `-lglut -lGLU -lGL -lX11 -lXmu`. `X11` und `Xmu` sind nötig da GLUT unter UNIX-Systemen auf X-Windows aufbaut.

Im Grundstudiumspool müssen außerdem noch Pfade zu den `.h`-Dateien und den OpenGL-Bibliotheken angegeben werden. Um `GL/glut.h` `#include`n zu können, wird der Pfad mit `-I/usr/X11R6/include` gesetzt (`-I` wie „include“), für die OpenGL-Bibliotheken mit `-L/usr/X11R6/lib` *vor* den `-l` Schaltern (`-L` wie „libraries“).

3.3 Wie sieht ein OpenGL-Programm aus?

Hier mal ein ganz einfaches OpenGL-Programm, dass zwei Punkte setzt:

```
/* Includes */

#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>

/* Funktionsprototypen */

void malen(void);

/* Funktionsdefitionen */

int main(int argc, char **argv)
{
   /* Initialisierung */

   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE |
      GLUT_RGB);
   /* Fensterbreite und -hoehe */
   glutInitWindowSize(400, 300);
   /* Fensterposition setzen */
   glutInitWindowPosition(100, 100);
   /* Fenstertitel setzen */
   glutCreateWindow("Hallo!");
   /* Hintergrundfarbe setzen */
   glClearColor(0.0, 0.0, 0.0, 0.0);
   /* Koordinatensystem setzen */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 400.0, 0.0, 300.0,
      -1.0, 1.0);
   /* Zeichnende Funktion setzen */
   glutDisplayFunc(&malen);
   /* Aktivieren des Fensters */
   glutMainLoop();
   return(0);
}

void malen(void)
   /* malt in das Fenster */
{
   /* Fenster loeschen */
   glClear(GL_COLOR_BUFFER_BIT);
   /* Punkte setzen */
   glBegin(GL_POINTS);
      /* Farbe angeben */
      glColor3f(1.0, 1.0, 1.0);
      /* Koordinaten angeben */
      glVertex2i(150, 150);
      /* Farbe angeben */
      glColor3f(1.0, 0.0, 0.0);
      /* Koordinaten angeben */
      glVertex2i(250, 150);
   glEnd();

   /* alle Kommandos ausfuehren */
   glFlush();
}
```

Hier werden nur zwei Punkte gezeichnet. Angenommen der Text steht in einer `.c`-Datei namens `main.c`, dann lässt sich das Programm im Grundstudiumspool mit der Kommando-Zeile (die hier leider zu breit für eine Zeile ist)

```
gcc main.c -I/usr/X11R6/include
-L/usr/X11R6/lib -lglut -lGLU -lGL
-lX11 -lXmu
```

compilieren und anschließend als `./a.out` aufrufen. Im Folgenden soll das Programm genauer analysiert werden:

Wie erwähnt müssen mit

```
#include <GL/glut.h>
```

die Funktionen der GLUT-, OpenGL- und GLU-Bibliotheken deklariert werden. Aus welcher Bibliothek eine Funktion kommt, lässt sich leicht an den Namen ablesen: `glut`... steht für GLUT, `gl`... für OpenGL und `glu`... für GLU, letzteres kommt hier nicht vor.

Die `main`-Funktion ist nur dazu da, um das ganze System zu initialisieren, ein OpenGL-Fenster zu öffnen und diese für das Zeichnen vorzubereiten.

Die GLUT-Funktionen `glutInit()`, `glutInitWindowSize()`, `glutInitWindowPosition()`, `glutCreateWindow()` und `glutMainLoop()` werden später behandelt, bzw. sind fast selbsterklärend. Zur Funktion `glutDisplayFunc()` sollte angemerkt werden, dass sie eine Funktionsadresse vom Typ `void (*)(void)` erwartet, die hier

einfach mit `&malen` als Adresse der Funktionsdefinition von `malen()` angegeben wird. Durch diese Funktionsadresse kann GLUT innerhalb der Funktion `glutMainLoop()` immer dann die Funktion `malen()` aufrufen, wenn der Fensterinhalt neu gezeichnet werden muss. `glutMainLoop()` ruft zu diesem Zweck gewissermaßen unser Programm zurück, weswegen Funktionen wie `malen()` auch *callback-Funktionen* genannt werden.

Die OpenGL-Kommandos `glMatrixMode()`, `glLoadIdentity()` und `glOrtho()` legen das verwendete Koordinatensystem bzw. die verwendete „Kameraposition und -richtung“ fest. Diese Möglichkeiten sind insbesondere für dreidimensionale Graphiken von besonderem Interesse, um zum Beispiel eine Kamerafahrt um ein Objekt herum zu simulieren. Da in diesem Kurs aber nur zweidimensionales OpenGL benutzt wird, brauchen wir diese Funktionen nicht weiter zu betrachten. Zu erwähen ist nur, das in diesem Beispiel über `glOrtho` das zweidimensionales Koordinatensystem so gesetzt wird, dass der Ursprung in der linken, unteren Ecke des Fensters liegt und die Koordinaten einfach die Position der Bildpunkte (Pixel) im Fenster angeben. Das hat den Vorteil, dass wir später die Koordinaten von Bildpunkte mit `int`s angeben können.

Die Funktion `glClearColor()` setzt die Rot-, Grün- und Blau-Anteile der Hintergrundfarbe jeweils von `0.0` (völlig dunkel) bis `1.0` (so hell wie möglich). Außerdem wird der α-Kanal der Hintergrundfarbe gesetzt, der aber in den allermeisten Anwendungen bedeutungslos ist. (Für Farben von Objekten im Vordergrund gibt der α-Kanal meistens die Opazität oder Undurchsichtigkeit an von `0.0` (durchsichtig, transparent) bis `1.0` (undurchsichtig, opaque).) Die Hintergrundfarbe wird in der Funktion `malen()` von `glClear(GL_COLOR_BUFFER_BIT);` benutzt, womit der gesamte Fensterinhalt mit der angegebenen Hintergrundfarbe gefüllt wird.

`GL_COLOR_BUFFER_BIT` ist eine von OpenGL definierte Konstante. Genauer gesagt ist es eine Zweierpotenz, also eine Ganzzahl mit nur einem gesetzten Bit. Solche Bits werden auch „flags“ genannt, also Flaggen, die bestimmte ja/nein-Aussagen signalisieren sollen. Zweierpotenzen können mit dem bitweisen ODER-Operator `|` kombiniert werden. Ein Beispiel dazu ist in

```
glutInitDisplayMode(GLUT_SINGLE |
   GLUT_RGB);
```

zu sehen. Hier werden die Konstanten `GLUT_SINGLE` und `GLUT_RGB` verODERt. Der Vorteil dieser Technik ist, dass mehrere ja/nein-Werte (Boolsche Werte) in einem `int` übergeben werden können und nicht eine ganze Reihe von Argumenten übergeben werden muss. Zum Beispiel erlaubt `glClear()` durch VerODERung mehrerer Bits einige weitere Buffer zu löschen. Aber wenn nur ein Buffer gelöscht werden soll, wie hier der Farbbuffer, dann reicht es, die jeweilige Bit-Flagge (hier `GL_COLOR_BUFFER_BIT`) anzugeben.

Das eigentliche Zeichnen des Fensterinhalts findet in der Funktion `malen()` statt. Hier wird zunächst mit `glClear()` der Fensterinhalt gelöscht und dann Objekte gezeichnet. `glBegin(GL_POINTS)` und `glEnd()` rahmen die Kommandos zum Zeichnen ein und geben an, dass Punkte gezeichnet werden sollen, andere graphische *Primitive* sind zum Beispiel Linien und Dreiecke. Diese Primitive können keine komplizierten geometrischen Objekte sein, weil sie von der Graphik-Hardware direkt dargestellt werden müssen.

Mit `glColor3f(1.0, 0.0, 0.0);` werden die Farben (Rot-, Grün- und Blau-Anteil) und mit `glVertex2i(250, 150);` die Koordinaten der Punkte gesetzt.

Das `glFlush();` Kommando stellt sicher, dass alle OpenGL-Kommandos ausgeführt und nicht länger gepuffert werden.

Beispiel: Farbverlauf

Um das Fenster mit anderen Inhalten zu füllen, muss nur die `malen()`-Funktion geändert werden. Folgende Funktion geht in zwei geschachtelten Schleifen für jede Koordinate `x` (jede Spalte) über alle Koordinaten `y` (alle Pixel in der Spalte `x`) und setzt den Pixel in einer Farbe, deren Rotanteil mit der `x`-Koordinate wächst und entsprechend der Grünanteil mit der `y`-Koordinate.

```
void malen(void)
{
   int x;
   int y;

   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_POINTS);
   x = 0;
   while (x < 400)
   {
      y = 0;
      while (y < 300)
```

```
        {
           glColor3f(x / 300.0,
              y / 400.0, 1.0);
           glVertex2i(x, y);
           y = y + 1;
        }
        x = x + 1;
     }
     glEnd();
     glFlush();
  }
```

3.4 Syntax von OpenGL-Kommandos

In dem Beispiel wurden schon einige syntaktische und lexikalische Eigenschaften von OpenGL-Kommandos deutlich: OpenGL-Befehle beginnen mit `gl`..., woran sich direkt weitere groß geschriebene Wörter anschließen. Das erste Wort ist oft ein Verb (wenn nicht, ist meistens `Set` gemeint) und beschreibt, welche Aktionen das Kommando durchführt. Konstanten beginnen mit `GL_`... und sind durchgängig groß geschrieben mit _ zwischen den Wörtern.

Einige Kommandos haben zusätzliche, angehängte Buchstaben wie `3f` und `2i` in `glColor3f()` bzw. `glVertex2i()`. Diese Buchstaben beschreiben nicht das Kommando, sondern die Anzahl (hier `3` bzw. `2`) und den Typ (hier `f` für Fließkomma- und `i` für Ganzzahlen) der Argumente. Viele OpenGL-Kommandos sind in verschiedenen Versionen mit unterschiedlichen Argumentanzahlen und -typen definiert, zum Beispiel gibt es auch eine `glColor4f()`-Funktion, die zusätzlich einen Wert für die Opazität erwartet, während `glColor3f()` die Opazität einfach auf `1.0` setzt. Tabelle 3.1 zeigt eine Übersicht der Buchstabenkürzel für Argumenttypen und die entsprechenden C- bzw. OpenGL-Typen.

Entsprechend könnte statt

```
glVertex2i(150, 150);
```

auch

```
glVertex2f(150.0, 150.0);
```

stehen, mit dem einzigen Unterschied, dass die Argumente einmal als Ganzzahlen und im zweiten Fall als Fließkommazahlen übergeben werden. Eine weitere gleichbedeutende Variante ist

	C-Typ	OpenGL-Typ
b	`signed char`	`GLbyte`
s	`short`	`GLshort`
i	`int` oder `long`	`GLint`, `GLsizei`
f	`float`	`GLfloat`, `GLclampf`
d	`double`	`GLdouble`, `GLclampd`
ub	`unsigned char`	`GLubyte`, `GLboolean`
us	`unsigned short`	`GLushort`
ui	`unsigned int` oder `unsigned long`	`GLuint`, `GLenum`, `GLbitfield`

Tabelle 3.1: Die OpenGL-Buchstabenkürzel für Argumenttypen.

```
glVertex3i(150, 150 , 0);
```

Diese Variante zeigt auch, dass die dritte Koordinate (z) eines Vertex von `glVertex2i()` implizit auf `0` gesetzt wird.

Von einigen OpenGL-Kommandos gibt es auch eine Vektor-Version, die durch ein angehängtes `v` kenntlich gemacht wird, und einen Vektor (Array, also: Adresse) von Werten als Argument erwarten statt mehrere Argumente. Z.B. könnte statt

```
glColor3f(1.0, 0.0, 0.0);
```

auch geschrieben werden

```
GLfloat farben[] = {1.0, 0.0, 0.0};

glColor3fv(farben);
```

Die letzte Zeile ist wieder äquivalent zu

```
glColor3fv(&farben[0]);
```

Wenn sich die OpenGL-Dokumentation auf alle Versionen eines OpenGL-Kommandos wie `glColor` beziehen will, wird oft z.B. `glColor*()` verwendet, wobei der `*` als Joker klarmachen soll, dass hier noch die richtigen Kürzel, je nach Argumenttyp ergänzt werden müssen. Entsprechend bezieht sich `glColor*v()` auf alle Vektor-Versionen von `glColor`.

3.5 OpenGL als Zustandsautomat

OpenGL ist ein Zustandsautomat, d.h. durch Befehle wird es in bestimmte Zustände versetzt, die solange aktiv bleiben, bis sie wieder durch Befehle geändert werden. Wie oben gezeigt, ist die aktive Farbe so eine Zustandsvariable. Nachdem mit `glColor*()` eine Farbe gesetzt wurde, bleibt sie aktiv bis sie durch eine andere Farbe ersetzt wird. D.h. bis dahin werden alle Objekte in dieser Farbe gezeichnet. Die aktive Farbe ist aber nur eine von vielen Zustandsvariablen, andere bestimmen die Blickrichtung, die Projektionstransformation, Linienstil, Position der Lichtquellen, Materialeigenschaften, etc. Viele Zustandsvariablen können aber auch nur mit `glEnable()` und `glDisable()` ein- bzw. ausgeschaltet werden.

Jede Zustandsvariable hat einen Default-Wert. Der aktuell aktive Wert der Variable kann mit `glGetBooleanv()`, `glGetDoublev()`, `glGetFloatv()`, `glGetIntegerv()`, `glGetPointerv()` oder `glIsEnabled()` abgefragt werden. Einige andere Zustandsvariablen besitzen weitere Abfragen, wie z.B. `glGetError()`. Darüberhinaus können Sätze von Zustandvariablen mit `glPushAttrib()` und `glPushClientAttrib()` auf einen Stack gerettet werden und von dort mit `glPopAttrib()` und `glPopClientAttrib()` wieder aktiviert werden.

3.6 Rendering-Pipeline

OpenGL verarbeitet Befehle in einer Pipeline, hier nur die wichtigsten englischen Schlagwörter:

- display lists (können OpenGL-Kommandos speichern und bei Bedarf wieder abspielen; sie sind eine Alternative zum „immediate mode“ in dem jedes OpenGL-Kommando sofort ausgeführt wird),
- evaluators (dienen zum Berechnen von komplexeren Objekten wie gekrümmten Flächen),
- per-vertex operations (sind alle Berechnungen, die für jeden Vertex durchgeführt werden, z.B. Transformation, Beleuchtung, Materialberechnungen, etc.),
- primitive assembly (darunter laufen einige Operationen auf den Primitiven (Punkte, Linien, Dreiecke,...) wie z.B. Clipping),
- pixel operations (Bilder, z.B. Texturen, nehmen einen etwas anderen Weg in der OpenGL-Pipeline, der unter diesem Begriff läuft),
- texture assembly (Texturierung von Primitiven mit Bildern ist ein wesentliches Element von Computergraphik; texture assembly organisiert die Texturen),
- rasterization (beschreibt die Umwandlung eines geometrischen Primitivs in eine Menge von Pixeln),
- fragment operations (sind Operationen auf Fragmenten, wie Texturierung und verschiedene Tests. Bevor ein Pixel tatsächlich gesetzt wird, heißt die Farbe für ein Pixel nur „Fragment“).

3.7 Verwandte Bibliotheken

Neben der schon erwähnten OpenGL Utility Library (GLU, Präfix `glu`), auf die in diesem Kurs nicht weiter eingegangen wird, und dem OpenGL Utility Toolkit (GLUT, Präfix `glut`), das später noch genauer vorgestellt wird, gibt es noch einige Plattform-abhängige Bibliotheken wie die OpenGL Extension to the X Window System (GLX, Präfix `glX`), für Windows 95/98/NT das Windows to OpenGL interface (WGL, Präfix `wgl`), für IBM OS/2 das Presentation Manager to OpenGL interface (PGL, Präfix `pgl`) und für Apple das AGL (Präfix `agl`).

GLUT erlaubt es zwar, einfache OpenGL-Anwendungen Plattform-unabhängig zu programmieren. Aber da es nicht mehr als der kleinste gemeinsame Nenner der verschiedenen Plattform-abhängigen Schnittstellen ist, muss für vollausgebaute OpenGL-Applikationen meistens doch auf eine Plattform-abhängige Bibliothek zurückgegriffen werden. In diesem Kurs begnügen wir uns aber mit GLUT.

Übungen zum zweiten Tag

Aufgabe II.1

Zeichenketten (Strings) werden in C bekanntermaßen als null-terminierte char-Arrays implementiert (siehe Abschnitt 2.2.2). In dieser Aufgabe sollen ein paar grundlegende String-Operation implementiert werden. Schreiben sie drei möglichst effiziente Funktionen, die die folgenden Aufgaben lösen: Die Länge eines Strings berechnen, zwei gegebene Strings zu einem neuen String verbinden und das Vorhandensein eines Teilstrings innerhalb eines zweiten Strings uberprüfen. Schreiben Sie ein Programm das zwei Zeichenketten als Kommandozeilenargumente erwartet und das Ergebnis der drei vorgenannten Operation auf die übergebenen Strings ausgibt.

Aufgabe II.2

Diese Aufgabe baut auf der Lösung von Aufgabe I.3 vom Vortag auf.

Schreiben Sie nun ein OpenGL-Programm, dass ein Diagramm folgender Art zeichnet: Die horizontale Achse soll Werte von 0.0 bis 4.0 von a (siehe Aufgabe I.2) und die vertikale Achse soll Werte von 0.0 bis 1.0 von f_n (siehe Aufgabe I.2) repräsentieren.

Zeichnen Sie in dieses Diagramm die Punkte mit Koordinaten (a, f_{100}), (a, f_{101}), (a, f_{102}), ..., (a, f_{199}) ein und zwar jeweils für a von 0.0 bis 4.0 mit einer geeigneten Schrittweiten.

Diese Art von Diagramm wird auch Feigenbaum-Diagramm (nach ihrem Erfinder Mitchel Feigenbaum) genannt. Versuchen Sie sich klarzumachen, was die dargestellten Strukturen bedeuten.

Aufgabe II.3

Die berühmten von Benoit B. Mandelbrot entdeckten Apfelmännchen beruhen auf einer zweidimensionalen iterativen Berechnungsvorschrift:

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$
$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n^2 - y_n^2 + c_x \\ 2x_n y_n + c_y \end{pmatrix},$$

Diese Iteration wird für *jeden* Bildpunkt durchgeführt bis $x_n^2 + y_n^2 > 4$ oder $n = 100$ (bzw. noch größer) ist. Die Koordinaten des Bildpunktes entscheiden dabei über die Konstanten c_x und c_y. Achten Sie bei der Implementierung darauf, dass die Skalierung und Verschiebung der Koordinaten so gewählt wird, dass sich für c_x Werte von -2.3 bis 1 und für c_y Werte von -1.25 bis 1.25 ergeben. Die Punktfarbe ergibt sich aus dem n, bei dem die Iteration für den jeweiligen Punkt abgebrochen wird. Traditionellerweise werden aber Punkte, für die der maximale Wert von n erreicht wird, schwarz gezeichnet. Die Menge dieser schwarzen Punkte ist dann eine Näherung der Mandelbrotmenge, also derjenigen Punkte, für die die Iteration niemals $x_n^2 + y_n^2 > 4$ erreicht.

Aufgabe II.4

Um Fensterinhalte wie in Aufgabe II.3 und nicht immer wieder neu berechnen zu müssen, speichert man die Bilddaten oft in einem Array zwischen und kopiert sie dann bei Bedarf aus diesem Array. Modifizieren Sie dazu Ihre Lösung von Aufgabe II.3 so, dass die berechneten Farben nicht direkt als Punkte auf dem Bildschirm ausgegeben werden, sondern zunächst in einem Array gespeichert werden. Zur Ausgabe auf dem Bildschirm sollen dann die Daten des Arrays als Punkte dargestellt werden.

Als Beispiel für elementare Bildverarbeitung weisen Sie vor der Ausgabe außerdem jedem Bildpunkt des Arrays (außer den Randpunkten) den Mittelwert aus seiner Farbe und der Farbe seiner vier Nachbarn zu. (Diese Operation kann auch mehrfach angewendet werden und führt zu immer „verwascheneren“ Bildern, d.h. hohe Frequenzen des Farbsignals werden herausgefiltert. Deswegen ist diese Operation auch ein einfaches Beispiel für einen Tiefpassfilter.)

Erweitern sie ihr Programm so, dass die Fenstergröße dynamisch über Kommandozeilenargumente angegeben werden kann. Die Größe in x-Richtung soll dabei über die Option `-x Pixelanzahl`, in y-Richtung über `-y Pixelanzahl` angegeben werden. Wenn beide oder einer der beiden Parameter fehlen, soll jeweils ein Defaultwert gesetzt werden. Da die Fenstergröße nicht mehr fest ist, muss für das Array zum Zwischenspeichern jetzt dynamisch Speicher allokiert werden.

(Tipp: Wandeln Sie die Angabe der Pixelanzahl Zeichen für Zeichen mit Hilfe des ASCII-Codes der Zeichen `['0'-'9']` in eine Integer-Zahl um.)

Teil III

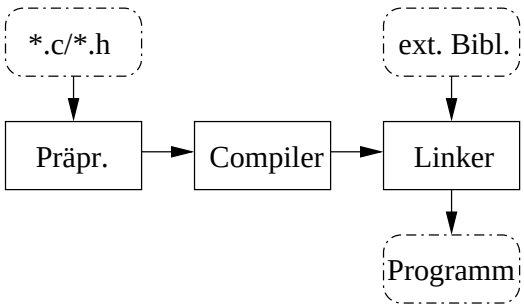
Dritter Tag

Kapitel 4

Der C-Übersetzungsprozess

Bisher waren die Beispiele auf einzelne `.c`-Dateien beschränkt mit eventuell mehreren Funktionsdefinitionen. Komplexe C-Programme bestehen aber meistens aus mehreren Dateien. Der Übersetzungsprozess, der diese C-Quelldateien in ein einzelnes ausführbares Programm überführt, verläuft dabei in mehreren Phasen.

Zunächst liest ein Präprozessor die Quelltexte und führt einige Textersetzungen aus. Das Ergebnis dieser Textersetzungen wird an den Compiler weitergereicht. Dieser erzeugt für die Eingabedatei eine Objektdatei mit Maschinencode. Greift diese Objektdatei auf Funktionlität anderer Objektdateien zu, muss sie mit den anderen Objektdateien gebunden werden. Dies erfolgt durch den Linker. Das Ergebnis schließlich ist das ausführbare Programm. Die folgende Abbildung gibt einen Überblick über die durchlaufenen Phasen:



Da die Aufgabe des Compilers bereits bekannt ist, sollen im Folgenden noch die Aufgaben des Präprozessors und des Linkers genauer beleuchtet werden.

4.1 Der Präprozessor

4.1.1 Was ist ein Präprozessor?

Bevor der eigentliche C-Compiler den Text einer `.c`-Datei auch nur anschaut, läuft der sogenannte Präprozessor über den Text. Der Präprozessor ist ein relativ schlichtes Gemüt und kann vor allem Text aus anderen Dateien einfügen, Textteile ersetzen und weglassen. Der Präprozessor kennt nur wenige Kommandos, die alle mit einem # beginnen. Die wichtigsten werden im Folgenden beschrieben.

4.1.2 #include

Mit Hilfe von

```
#include <Filename>
```

oder

```
#include "Filename"
```

kann an der Stelle dieses Kommandos der Inhalt der Datei namens *Filename* eingefügt werden. Im Allgemeinen werden damit nur `.h`-Dateien `#included`.

4.1.3 #define und #undef

Mit

```
#define Symbolname Ersetzungstext
```

wird der Präprozessor angewiesen, im nachfolgenden Text *Symbolname* durch *Ersetzungstext* zu ersetzen. `#define` wird vor allem genutzt, um Konstanten einen Namen zu geben, z. B. kann mit

```
#define PI 3.1415927
```

erreicht werden, dass vor der Compilierung `PI` jeweils durch `3.14151927` ersetzt wird. Der *Symbolname* für solche Konstanten besteht meistens nur aus Großbuchstaben. Ein Beispiel, das meist in `stddef.h` vereinbart wird, ist `NULL` für `0`.

Der *Ersetzungstext* darf auch fehlen, also „leer“ sein.

Das Gegenstück zu `#define` ist `#undef` *Symbolname* und macht ein vorheriges `#define` dieses *Symbolname*ns unwirksam.

4.1.4 Makros

`#define` ist aber noch mächtiger, weil auch „Argumente“ benutzt werden können. `#define`s mit Argumenten werden auch Makros genannt. Zum Beispiel:

```
#define betrag(a) ((a) >= 0 ? \
   (a) : -(a))
```

Falls eine Makrodefinition (wie diese) mehr als eine Zeile lang werden sollte, muss vor dem Ende der Zeile ein \ stehen, um dem Präprozessor mitzuteilen, dass die nächste Zeile auch noch zur Makrodefinition gehört.

Wenn nach dieser „Makrodefinition“ (ein ungeschickter Begriff, weil es keine Definition im Sinne von C ist) im Programmtext z. B.

```
betrag(100. * x)
```

vorkommt, wird dieser Ausdruck durch das Makro ersetzt, wobei das `a` aus der Makrodefinition durch das Argument `100. * x` ersetzt wird. Das Ergebnis ist also

```
((100. * x) >= 0 ? (100. * x) :
   -(100. * x))
```

Falls also `100. * x` größer oder gleich `0` ist, ergibt sich `100. * x` sonst der Wert `-(100. * x)`, der dann größer `0` ist.

Die vielen Klammern in der Makrodefinition sind notwendig, weil wir nicht wissen welche Operatoren in dem Makroargument auftauchen. Beispiel: Wir benutzen die Makrodefinition

```
#define betrag(a) a >= 0 ? a : -a
```

und verwenden später

```
5 * betrag(a - b)
```

Daraus macht der Präprozessor dann

```
5 * a - b >= 0 ? a - b : -a - b
```

Das funktioniert aus zwei Gründen nicht: Die `5` multipliziert nicht das Ergebnis sondern nur das `a` in dem Bedingungsausdruck und das zweite Ergebnis `-a - b` ist etwas anderes als das gewünschte `-(a - b)`. Es ist also nötig in Makrodefinitionen den Argumentnamen im *Ersetzungstext* immer zu klammern und zusätzlich den gesamten *Ersetzungstext* einzuklammern.

4.1.5 #if

Der Präprozessor kann angewiesen werden, Text unter der Bedingung einzufügen, dass ein bestimmter *Symbolname* `#define`d wurde oder auch, dass ein bestimmter *Symbolname nicht* `#define`d wurde. Zum Beispiel:

```
#if defined DEBUGGING
   printf("Hallo!\n");
#endif
```

Der Text zwischen der `#if`- und `#endif`-Anweisung wird nur dann vom Präprozessor durchgelassen, wenn der *Symbolname* `DEBUGGING` vorher mit `#define` vereinbart wurde. Ansonsten wird der Text (hier also der Aufruf von `printf()`) einfach weggelassen und deshalb auch nicht compiliert.

`#if defined` ist äquivalent zu `#ifdef`. Das Gegenstück ist `#if !defined` *Symbolname* bzw. `#ifndef` *Symbolname*, dabei wird der Text bis zum `#endif` nur dann weitergereicht wenn der *Symbolname* nicht `#define`d wurde.

In dieser Familie gibt es zusätzlich noch die Präpozessor-Anweisung `#else`, die zusammen mit `#if` genauso funktioniert, wie `if` und `else` in der eigentlichen Sprache C.

Anders als Kommentare können `#if`-`#endif`-Blöcke auch geschachtelt werden. Sie eignen sich deshalb auch zum „Auskommentieren“ von (kommentierten) Programmteilen. Einrücken von Präprozessor-Anweisungen ist aber problematisch, da ältere C-Compiler verlangen, dass das # am Anfang der Zeile steht. Nach dem # dürfen aber meistens noch Leerzeichen stehen.

Hier noch ein Beispiel: Oft wird `TRUE` als `1` und `FALSE` als `0` definiert. Manchmal ist aber nicht klar, ob diese Symbole schon in einer `#include`ten `.h`-Datei vereinbart wurden. Mit den folgenden Zeilen kann dann eine doppelte Vereinbarung desselben Symbols vermieden werden:

```
#if !defined TRUE
```

```
#  define TRUE 1
#endif
#if !defined FALSE
#  define FALSE 0
#endif
```

Doppelte Vereinbarungen und Deklarationen können unter Umständen unangenehme Konsequenzen haben, das trifft natürlich um so mehr auf doppelt `#include`te `.h`-Dateien zu. Eine `.h`-Datei kann sich aber dagegen schützen, mehr als einmal `#include`d zu werden. Zum Beispiel kann eine Datei namens `meins.h` mit den Zeilen

```
#if !defined MEINS_H
#define MEINS_H
```

beginnen und den `#if`-`#endif`-Block ganz am Ende mit

```
#endif
```

schließen. Dadurch wird der Inhalt der Datei nur dann vom Präprozessor weitergereicht, wenn das Symbol `MEINS_H` noch nicht definiert wurde, also normalerweise beim ersten `#include` von `meins.h`.

4.1.6 Der ##-Operator

Mit Hilfe des ##-Operators kann ein neuer *Bezeichner* (z. B. Variablen-, Funktions- oder Typname) mit Hilfe eines Makros aus zwei Teilen zusammengesetzt werden. Zum Beispiel:

```
#define verbinde(a,b) a##b
```

Folgt nun im Programmtext zum Beispiel

```
int verbinde(ich, bin);
```

dann wird daraus

```
int ichbin;
```

Der ##-Operator wird sehr selten eingesetzt.

4.2 Der Linker

(Dieser Abschnitt ist eine Art überarbeiteter Auszug aus Kapitel 4 des Buchs „Der C-Experte“ von Andrew Koenig.)

4.2.1 Was ist ein Linker?

Ein C-Compiler übersetzt in der Regel nur einzelne `.c`-Dateien in Maschinenbefehle. Das hat für große Programme den ungeheuren Vorteil, dass nicht alle `.c`-Dateien neu übersetzt werden müssen, wenn nur eine einzelne `.c`-Datei geändert wurde und deshalb neu compiliert werden muss. Außerdem müssen Bibliotheksfunktionen (die ja meistens auch in C programmiert sind) nicht jedesmal neu übersetzt werden.

Wie lassen sich trotz dieser *getrennten Compilierung* verschiedene `.c`-Dateien und Funktionen aus Bibliotheken zu einem Programm zusammenfassen? Im Wesentlichen ist das die Aufgabe eines Programms, das Linker genannt wird. Als Aufgabe des C-Compilers ergibt sich dann, `.c`-Dateien so zu übersetzen, dass der Linker sie verwenden kann.

In den Beispielen bisher wurde der Linker immer automatisch vom Compiler aufgerufen, der nur eine `.c`-Datei übersetzen musste. Dem Compiler können aber auch mehrere `.c`-Dateien zum Übersetzen gegeben werden. Z. B. eine Datei namens `main.c` in der stehen könnte:

```
#include <stdio.h>

int verdopple(int a);

int main(void)
{
   printf("%d\n", verdopple(100));
   return (0);
}
```

und eine andere Datei namens `verdopple.c`, in der die Definition der Funktion `verdopple()` steht:

```
int verdopple(int a)
{
   return(2 * a);
}
```

Das Programm aus diesen beiden kleinen `.c`-Dateien kann dann z. B. mit

```
gcc main.c verdopple.c
```

erzeugt werden. Dabei übersetzt der Compiler die beiden `.c`-Dateien getrennt und der Linker verbindet die übersetzten Dateien zum Programm `a.out`. (Einzelne Funktionsdefinitionen können nicht über mehrere Dateien verteilt werden, sondern jede `.c`-Datei enthält

wie in dem Beispiel nur vollständige Funktionsdefinitionen.)

Normalerweise erzeugt der Compiler aus einer `.c`-Datei ein Objektmodul (Dateiendung: `.o`). Solche Objektmodule fasst der Linker zu einem ausführbaren Programm zusammen. Dabei benutzt er manche Objektmodule direkt als `.o`-Datei, andere holt er aus Bibliotheken von Objektmodulen, z. B. aus den Standardbibliotheken oder anderen Bibliotheken, z. B. der von OpenGL. Entsprechend wird jetzt auch klar, dass die Kommandozeilenbefehle `-L` und `-l` des C-Compilers direkt an den Linker weitergegeben werden, damit der Linker die notwendigen Bibliotheken findet.

Ein Linker betrachtet ein Objektmodul als eine Menge *externer Objekte*. Jedes externe Objekt entspricht dem Inhalt eines Teils des Speichers und wird über einen *externen Namen* angesprochen. Jede Funktion, die nicht als `static` deklariert wurde, ist ein externes Objekt. Das gilt auch für externe Variablen, die nicht als `static` deklariert wurden. Beispiel: Wenn in der Datei `verdopple.c` stehen würde

```
static int verdopple(int a)
{
   return(2 * a);
}
```

dann wäre die Funktion `verdopple()` nur innerhalb der Datei `verdopple.c` sichtbar, d.h. sie darf *nicht* aus der `main()`-Funktion in `main.c` aufgerufen werden. Allerdings kann dieser Fehler dem Compiler nicht auffallen, da er immer nur einzelne Dateien betrachtet und in diesem Fall ist weder in `main.c` noch in `verdopple.c` ein Fehler. Dem Linker wird dieser Fehler aber auffallen, und er wird eine Fehlermeldung ausgeben.

Externe Variablen, sind Variablen, die außerhalb von Funktionsdefinitionen deklariert (oder definiert) werden. Externe Variablen werden aber nur dann zu externen Objekten (und heißen dann *globale* Variablen), wenn sie nicht als `static` deklariert wurden. Beispiele dazu sind in den nächsten Abschnitten.

Ein Linker funktioniert also in etwa so: Er bekommt eine Reihe von Objektmodulen und Bibliotheken als Eingabe und produziert daraus als Ausgabe ein ausführbares Programm.

Im Allgemeinen sollten in einem ausführbaren Programm alle externen Objekte, insbesondere Funktionen verschiedene Namen besitzen, damit es nicht zu Zweideutigkeiten bei der Verwendung eines Namens kommt. Das würde insbesondere dann zu Problemen führen, wenn ein Objektmodul eine Referenz auf ein externes Objekt enthält, das in mehreren anderen Objektmodulen unter dem gleichen Namen existiert. Normalerweise muss der Linker nur alle externen Objekte, die in den Objektmodulen definiert sind, sammeln und die Referenzen auf diese externen Objekte auflösen, d. h. ihre tatsächliche Adresse einsetzen.

4.2.2 extern

Eine Deklaration, die zur Reservierung oder Belegung von Speicherplatz führt, wird als Definition bezeichnet. Deshalb ist die Deklaration

```
int a;
```

außerhalb einer Funktionsdefinition die *Definition* des externen Objekts `a`. Sie deklariert `a` als externe Ganzzahlvariable und reserviert entsprechend viel Speicherplatz. Die Deklaration

```
int a = 42;
```

ist eine Definition von `a`, die den Anfangswert `42` vorgibt. Also wird nicht nur Speicherplatz reserviert, sondern auch der Anfangswert für diesen Speicherplatz angegeben. Variablen, die extern definiert, aber nicht initialisiert werden, bekommen zu Beginn des Programmablaufs den Wert `0`, bzw. `0.0`.

Das Keyword `extern` dient dazu externe Variablen in einem Modul zu deklarieren, die *nicht* in diesem Modul definiert werden, d.h. für die in diesem Modul kein Speicherplatz reserviert wird. Die Deklaration

```
extern int a;
```

ist also *keine* Definition von `a`. `a` wird zwar als eine externe Integervariable deklariert, aber durch Angabe des Keywords `extern` wird ausgesagt, dass `a` in einem anderen Modul definiert wird. Für den Linker ist diese Deklaration eine Referenz auf ein externes Objekt `a`. Deshalb hat sie auch innerhalb einer Funktionsdefinition die gleiche Bedeutung.

Da jedes externe Objekt eines ausführbaren Programms Speicherplatz benötigt (Variablen, um ihre Werte zu speichern, Funktionen, um die entsprechenden Maschinenbefehle zu speichern), muss jedes externe Objekt einmal definiert werden. Deshalb muss in einem Programm, in dem

```
extern int a;
```

steht, irgendwo auch

```
int a;
```

vorkommen, entweder in demselben Objektmodul oder einem anderen.

Was passiert, wenn diese Definition mehrmals vorkommt? Manche Compiler erlauben das und erwarten, dass der Linker die Objekte zu einem Objekt zusammenführt. Dann ergibt sich aber ein Problem, wenn das Objekt in den Definitionen unterschiedlich initialisiert wurde. Deshalb ist der Standard die Forderung, dass es genau eine Definition jedes externen Objekts gibt.

Für Funktionen gelten dieselben Regeln. Allerdings ist der Unterschied zwischen Funktionsdefinitionen und -deklarationen (Prototypen) an der An- oder Abwesenheit des Funktionsblocks zu erkennen. Das Keyword `extern` ist deshalb für Funktionen nicht nötig.

4.2.3 static

Um Namenskonflikte von externen Objekten zu vermeiden, können externe Variablen mit dem Modifizierer `static` definiert werden und sind dann nur noch innerhalb des Moduls sichtbar. Zum Beispiel bedeutet die Definition der externen Variable `a`

```
static int a;
```

innerhalb derselben `.c`-Datei dasselbe wie

```
int a;
```

mit dem Unterschied, dass `a` vor anderen Objektmodulen versteckt ist. Wenn sich also mehrere Funktionen dieselben externen Objekte miteinander teilen, dann sollten diese Funktionen in einer `.c`-Datei zusammengefasst werden und die benötigten Objekte als `static` definiert werden.

Wieder überträgt sich dieses Konzept auf Funktionen: Eine mit `static` definierte Funktion ist nur innerhalb derselben `.c`-Datei sichtbar. Die Definition einer Funktion, die nur von einer Funktion (oder wenigen anderen) aufgerufen wird, sollte daher mit den Definitionen der aufrufenden Funktionen in einer `.c`-Datei stehen und als `static` definiert werden.

Der Modifizierer `static` wurde bereits in Kapitel 1.5 in einem anderen Zusammenhang verwendet und zwar zum definieren *statischer* Variablen innerhalb von Funktionen. Diese werden nicht bei jedem Funktionsaufruf neu erzeugt und am Ende des Funktionsaufrufs wieder ungültig, sondern behalten ihren Wert auch zwischen Funktionsaufrufen bei.

4.2.4 Deklarationsdateien

... werden auch Header- oder kurz `.h`-Dateien genannt. Sie enthalten normalerweise die *Deklarationen* von externen Objekten, die in einer `.c`-Datei definiert werden, die bis auf die Endung den gleichen Namen hat. `.h`-Dateien von Bibliotheken enthalten oft die Deklarationen von mehreren `.c`-Dateien, und `#includen` weitere `.h`-Dateien.

Es ist sinnvoll, dass die `.h`-Datei von der entsprechenden `.c`-Datei `#included` wird, denn damit sind zumindest die externen Objekte alle deklariert und in der `.c`-Datei müssen keine weiteren Funktionsprototypen angegeben werden. `static`-Objekte sollten aber nicht in die `.h`-Datei aufgenommen werden. Vielmehr sollten die `static`-Funktionsprototypen am Anfang der `.c`-Datei aufgelistet sein.

Der eigentliche Sinn von `.h`-Dateien ist, dass sie von `.c`-Dateien `#included` werden, die Bezug auf externe Objekte anderer `.c`-Dateien nehmen, um so die korrekte Deklaration der externen Objekte zu gewährleisten. Daneben können aber auch alle Typ-, `struct`- und andere Deklarationen der `.h`-Datei benutzt werden.

4.3 make

4.3.1 Was ist make?

In den Beispielen wurde das Kommando zum Compilieren durch die Benutzung von OpenGL erheblich länger. Bei größeren C-Programmen aus vielen `.c`-Dateien und mehreren Bibliotheken ist das Eintippen solche Kommandos noch lästiger. Außerdem ist es sinnvoll den Vorteil des getrennten Compilierens von `.c`-Dateien auszunutzen: `.c`-Dateien müssen nur dann neu compiliert werden, wenn sie oder die `.h`-Dateien, die `#included` werden, geändert wurden. Die neuen Objektmodule können dann mit den unveränderten vom Linker zu einem Programm zusammengefasst werden. Bei einer großen Zahl von `.c`-Dateien kann das den Zeitaufwand fürs Compilieren ganz erheblich verringern.

`make` ist ein Kommando bzw. Teil einer C-Entwicklungsumgebung, das diese Idee unterstützt. Dazu muss ein *Makefile* angegeben werden, das üblicherweise `Makefile` oder `makefile` heißt. Wenn `make` aufgerufen wird, sucht es nach einer Datei namens `Makefile` oder `makefile` und baut ausgehend von den enthaltenen Informationen ein Programm neu auf.

4.3.2 Makefiles

Üblicherweise enthalten Makefiles im Wesentlichen diese Informationen:

- Den Namen des ausführbaren Programms,
- Dateinamen von `.c`-Dateien und `.o`-Dateien (Objektmodulen),
- Compiler- und Linker-Optionen,
- Beschreibungen der Abhängigkeiten,
- Kommandos zur Erzeugung des Programms, bzw. der Objektmodule.

(Nicht berücksichtigt ist dabei die Möglichkeit C-Bibliotheken zu erzeugen.) „Abhängigkeit“ einer Datei bedeutet, dass z.B. ein Objektmodul neu erzeugt werden muss, wenn sich die entsprechende `.c`-Datei seit dem letzten Aufruf von `make` geändert hat. Ebenso muss sie neu erzeugt werden, wenn eine der `#include`ten `.h`-Dateien geändert wurde. Die `.o`-Datei eines Objektmoduls ist also „abhängig“ von der entsprechenden `.c`-Datei und allen darin `#include`ten `.h`-Dateien.

Das ausführbare Programm wird vom Linker aus den `.o`-Dateien und den Bibliotheken erzeugt. Da sich Bibliotheken meistens nicht ändern (wie gehen davon aus, dass keine Bibliotheken neu erzeugt werden), ist die Datei des ausführbaren Programms nur von den `.o`-Dateien abhängig.

Die Syntax des Makefiles soll anhand des ersten OpenGL-Beispiels klargemacht werden. Damit der Umgang mit mehreren Dateien klar wird, ist das Programm in drei (unkommentierte) Dateien aufgeteilt. `main.c` sieht so aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>

#include "malen.h"

int main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE |
      GLUT_RGB);
   glutInitWindowSize(400, 300);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Hallo!");
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 400.0, 0.0, 300.0,
      -1.0, 1.0);
   glutDisplayFunc(&malen);
   glutMainLoop();
   return(0);
}
```

Hier steht wirklich nur noch die `main()`-Funktion. Die Funktion `malen()` wird beim Aufruf von `glutDisplayFunc()` referenziert und muss daher durch das `#include` von `malen.h` deklariert werden. (Hier wird `#include "malen.h"` statt `#include <malen.h>` verwendet, um deutlich zu machen, dass `malen.h` nicht aus einer fertigen Bibliothek kommt, sondern Teil des (sich änderenden) Programms ist.) `malen.h` sieht so aus:

```
#if !defined MALEN_H
#define MALEN_H

#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>

void malen(void);

#endif
```

Wie erwähnt dienen die ersten beiden und die letzte Zeile dazu, ein wiederholtes `#include`n zu verhindern. Nach der Bearbeitung durch den Präprozessor stehen hier nur noch Deklarationen, keine Definitionen. Die Definition von `malen()` findet sich schließlich in `malen.c`:

```
#include "malen.h"

void malen(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_POINTS);
      glColor3f(1.0, 1.0, 1.0);
      glVertex2i(150, 150);
      glColor3f(1.0, 0.0, 0.0);
      glVertex2i(250, 150);
   glEnd();
   glFlush();
}
```

Die erste Zeile dient nicht nur dazu die Funktion `malen()` zu deklarieren (was hier sowieso nicht notwendig ist), sondern ist notwendig um die `.h`-Dateien von OpenGL zu `#include`n.

Die Entscheidung, ob diese `.h`-Dateien in der `.c`-Datei oder der entsprechenden `.h`-Datei `#include`t werden sollen, ist hier nicht wichtig. Manchmal müssen sie aber in der `.h`-Datei stehen, weil sie Typen deklarieren, die auch in den Funktionsprototypen verwendet werden. Entsprechend sind z. B. auch `struct`-Deklarationen meistens nicht in einer `.c`-Datei, sondern in der entsprechenden `.h`-Datei.

Ein einfaches Makefile für diese Programm könnte nun so aussehen:

```
SRCS = main.c malen.c
OBJS = main.o malen.o
CC = gcc
CFLAGS = -g -ansi -Wall \
   -I/usr/X11R6/include
SYSLIBS = -L/usr/X11R6/lib -lglut \
   -lGLU -lGL -lX11 -lXmu -lm

main : $(OBJS)
   $(CC) $(OBJS) $(SYSLIBS) \
   $(CFLAGS) -o $@

main.o : malen.h

malen.o : malen.h

.c.o :
   $(CC) -c $(CFLAGS) $<
```

Überlange Zeilen sind hier durch ein \ getrennt, so wie bei mehrzeiligen Präprozessor-Anweisungen. Wichtig ist auch, dass die Einrückungen in Makefiles durch „harte“ Tabulatoren (also nicht durch eine Reihe von Leerzeichen, sondern durch ein Tabulatorzeichen) erzeugt werden müssen.

Dieses Makefile definiert zunächst einige Symbole: `SRCS` (für *sources*) als die Liste der `.c`-Dateien, `OBJS` (für *objects*) als die Liste der entsprechenden `.o`-Dateien, `CC` als das Kommando zum Aufruf des C-Compilers und `CFLAGS` als die Optionen des C-Compilers.

Dann werden mit

```
main : $(OBJS)
```

die Abhängigkeiten des ausführbaren Programms `main` angegeben. (Links von `:` steht das zu erzeugende *Ziel* (in diesem Fall die Datei `main`) und rechts davon die Dateien, von denen es abhängt.) Wie erwähnt hängt das Programm nur von den `.o`-Dateien ab, die in dem Symbol `OBJS` definiert wurden. (Mit `$(...)` wird in einem Makefile der Wert eines vorher im Makefile definierten Symbols eingesetzt.) In den nächsten Zeilen müssen die Anweisung stehen, um die Zieldatei (hier also das ausführbare Programm) zu erzeugen. Vor jeder Anweisung muss ein Tabulatorzeichen stehen, damit `make` die Zeile als Anweisung versteht. Werden in der Anweisung

```
$(CC) $(OBJS) $(SYSLIBS) \
$(CFLAGS) -o $@
```

alle Symbole eingesetzt, dann ergibt sich eine ganz ähnliche Kommandozeile, wie sie schon vorher in den Beispielen verwendet wurde. Das `$@` in `-o $@` bezieht sich auf den Namen der Zieldatei (hier `main`) und weist deshalb den C-Compiler an, das fertige Programm `main` zu nennen.

Die beiden Zeilen

```
main.o : malen.h

malen.o : malen.h
```

enthalten keine Anweisungen, sondern nur die schon erwähnten Abhängigkeiten von `.h`-Dateien.

Mit der Suffix-Regel

```
.c.o :
   $(CC) -c $(CFLAGS) $<
```

wird `make` angewiesen *jede* `.o`-Datei aus der gleichnamigen `.c`-Datei mit der Anweisung

```
$(CC) -c $(CFLAGS) $<
```

zu erzeugen. Das `-c` weist dabei den C-Compiler an, nur zu compilieren und nicht den Linker aufzurufen, deshalb müssen auch keine Bibliotheken angegeben werden. (`$<` steht für den Namen der jeweiligen `.c`-Datei.)

Mit Hilfe dieses Makefiles ist es nun möglich einfach `make main` (oder nur `make`, das ist äquivalent weil `main` die erste angegebene Zieldatei ist) aufzurufen. `make` findet dann automatisch heraus, welche `.c`-Dateien geändert wurden und deshalb neu übersetzt werden müssen.

Manchmal ist es aber notwendig das Projekt komplett neu zu übersetzen, z. B. wenn andere C-Compiler-Optionen verwendet werden sollen. Dazu müssen alle

.o-Dateien gelöscht werden, was auch einfach in ein Makefile eingebaut werden kann:

```
clean :
   rm $(OBJS)
```

Falls make clean eingegeben wird, werden mit Hilfe des rm Kommandos alle .o-Dateien gelöscht und müssen bei späteren make-Aufrufen gegebenenfalls neu erzeugt werden. Weil clean keine echte Datei ist, und damit es nicht zu einem Konflikt mit einer Datei namens clean kommt, sollte noch .PHONY : clean eingefügt werden.

Außerdem wird üblicherweise das erste Ziel all genannt und führt zur Erstellung aller Programme. In unserem Fall also nur von main. Die Zeile nach den Symboldefinitionen sollte deshalb heißen:

```
all : main
```

4.3.3 makedepend

Bei größeren C-Programmen mit sehr vielen .c- und .h-Dateien ist die Eingabe der Abhängigkeiten mit einem gewissen Aufwand verbunden. makedepend ist in der Lage, die Abhängigkeiten der .o-Dateien selbständig zu ermitteln und in das Makefile einzubauen. Zur Demonstration beginnen wir mit einem Makefile ohne die Abhängigkeiten:

```
SRCS = main.c malen.c
OBJS = main.o malen.o
CC = gcc
CFLAGS = -g -ansi -Wall \
   -I/usr/X11R6/include
SYSLIBS = -L/usr/X11R6/lib \
   -lglut -lGLU -lGL \
   -lX11 -lXmu -lm

all : main

main : $(OBJS)
   $(CC) $(OBJS) \
   $(SYSLIBS) $(CFLAGS) -o $@

.PHONY : clean
clean :
   rm $(OBJS)

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

depend :
   makedepend -Y $(SRCS)
```

Hier wurde ein Ziel depend angegeben, das zum Aufruf von makedepend führt und die .c-Dateien an makedepend übergibt. Mit make depend wird diese Anweisung ausgeführt. makedepend fügt dann am Ende des Makefiles z. B. die Zeilen:

```
# DO NOT DELETE

malen.o: malen.h
main.o: malen.h
```

ein, die die Abhängigkeiten der .o-Dateien enthalten.

Natürlich wird an diesen akademischen Beispielen der Nutzen von makedepend und vielleicht von Makefiles überhaupt noch nicht allzu deutlich. Beispiele aus dem „richtigen Leben“ sind aber nicht nur größer, sondern auch unübersichtlicher, so dass sie vielleicht für eine Einführung noch ungeeigneter sind.

Kapitel 5

Eigene Datentypen in C

Nachdem das Konzepts der Aufteilens eines Programms in mehrere Module und deren Zusammenühren mit Hilfe des Linkers nun bekannt ist, ist es uns jetzt möglich Programmcode zu schreiben, den wir immer wieder in verschiedenen Programmen verwenden können. Durch die Auslagerung in eigene Programmodule muss der Quelltext nicht kopiert werden, sondern kann durch einbinden der entsprechenden Header-Dateien und dazulinken der zugehörigen Objektmodule wiederverwendet werden. Diese Konzept hat den Vorteil, das eventuelle Änderungen und Fehlerbehebungen nur einmal vorgenommen werden müssen und für alle Programme, die die enstprechende Funktionalität verwenden, sofort verfügbar sind.

In diesem Abschnitt soll die Erzeugung von wiederverwendbarem Code am Beispiel der Implementierung eines neuen Datentyps verdeutlicht werden..

5.1 Gekapselte Datentypen in C

Als Beispiel für einen nützlichen Datentyp sollen hier dreidimensionale Vektoren betrachtet werden. Zunächst werden die Strukturelemente *(data members)* festgelegt. Für einen dreidimensionalen Vektor bieten sich 3 `double`s an. Wir deklarieren also eine `struct` mit 3 `double`s:

```
struct Vector3d
{
   double x;
   double y;
   double z;
};
```

Das Kürzel `3d` steht für 3 `double`s. Da der Datentyp `struct Vector3d` etwas umständlich zu schreiben ist, vereinbaren wir besser einen neuen Typ. Aus obiger `struct` Deklaration wird also:

```
typedef struct
{
   double x;
   double y;
   double z;
} Vector3d;
```

Der Datentyp heißt jetzt `Vector3d` (statt `struct Vector3d`), sonst hat sich nichts geändert.

Damit haben wir schon eine wichtige Entscheidung getroffen: Der Typ unseres Datentyps ist eine `struct`, *nicht* eine Adresse auf eine `struct`-Variable. Der Unterschied ist bei großen `struct`s ganz erheblich, denn um eine große `struct`-Variable als Argument an eine Funktion zu übergeben, muss einiges an Speicherplatz reserviert und kopiert werden. Die Adresse als Argument zu übergeben, ist für große `struct`s wesentlich schneller. Meistens werden deshalb Adressen übergeben. Hier sollen beide Möglichkeiten vorgestellt werden. Zunächst ist unsere `struct` mit 3 `double`s noch klein genug, um sie komplett übergeben zu können. Im nächsten Abschnitt wird die Alternative mit Adressen (Referenzen) vorgestellt.

Um Namenskonflikte zu verhindern, lassen wir die Namen aller Funktionen, die diesen Datentyp verarbeiten *(function members)*, mit `v3d_...` beginnen (ähnlich wie OpenGL seine Kommandos mit dem Prefix `gl...` kenntlich macht). Ausserdem schreiben wir alle unsere Funktionen zum Datentyp `Vector3d` in eine Datei `vector3d.c` mit einer `.h`-Datei `vector3d.h`.

Welche Funktionen brauchen wir? Neben den mathematischen Operationen für dreidimensionale Vektoren gibt es einige wichtige Funktionen, die für *jeden* so zusammengesetzten Datentyp sinnvoll sind:

- eine Funktion zur Erzeugung einer Variable von diesem Typ (der sogenannte *Konstruktor*, das Gegenstück (einen *Destruktor*) brauchen wir hier noch nicht),
- Funktionen zum Setzen und Auslesen der Strukturelemente (sogenannte *data access functions*).

Der zweite Punkt ist sehr wichtig zur Kapselung: Ein Programm-Modul, das unsere Funktionen verwendet, sollte nicht von der Implementation des Datentyps abhängig sein. D. h. wenn wir uns entscheiden sollten, unseren Vektor nicht mehr mit kartesischen Koordinaten zu speichern, sondern z. B. mit Kugelkoordinaten, dann sollte es ausreichen, unsere Funktionen entsprechend umzuschreiben. Alle anderen Programm-Module sollten davon nicht betroffen werden. Die Implementation sollte im Sinne des *information hiding* versteckt sind. In C ist es leider so, dass wir gezwungen sind, unsere Datentypdeklaration in der `.h`-Datei anzugeben, so dass jeder Nutzer (auch wenn er die `.c`-Datei nicht hat), Einblick auf die Strukturelemente haben kann. Echtes *information hiding* ist also in C nicht möglich: Wir müssen darauf vertrauen, dass Benutzer unserer Funktionen nie auf die Strukturelemente zugreifen und stattdessen unsere Zugriffsfunktionen benutzen. Damit diese Funktionen wenigstens nicht langsamer sind als der direkte Zugriff, verwenden wir Makros.

Neben den erwähnten Funktionen soll in unserem Beispiel nur die Länge des Vektors und der normalisierte Vektor berechnet werden. Damit könnte unsere `.h`-Datei so aussehen (auf Englisch, ohne Kommentare):

```
#if !defined VECTOR3D_H
#define VECTOR3D_H

#include <stdlib.h>
#include <math.h>

typedef struct
{
   double x;
   double y;
   double z;
} Vector3d;

#define v3d_get_x(v) ((v).x)
#define v3d_get_y(v) ((v).y)
#define v3d_get_z(v) ((v).z)

#define v3d_set_x(v, new_x) \
   ((v).x = (new_x))
#define v3d_set_y(v, new_y) \
   ((v).y = (new_y))
#define v3d_set_z(v, new_z) \
   ((v).z = (new_z))

Vector3d v3d_new(double x,
   double y, double z);

double v3d_get_length(Vector3d v);
Vector3d v3d_normalized(
   Vector3d v);

#endif
```

Und die entsprechende `.c`-Datei sieht so aus:

```
#include "vector3d.h"

Vector3d v3d_new(double x,
   double y, double z)
{
   Vector3d v;

   v.x = x;
   v.y = y;
   v.z = z;

   return(v);
}

double v3d_get_length(Vector3d v)
{
   return(sqrt(v.x * v.x +
      v.y * v.y + v.z * v.z));
}

Vector3d v3d_normalized(
   Vector3d v)
{
   Vector3d n;
   double length;

   length = v3d_get_length(v);

   if (length > 0.0)
   {
      n.x = v.x / length;
      n.y = v.y / length;
      n.z = v.z / length;
```

```
    }
    else
    {
       n.x = 0.0;
       n.y = 0.0;
       n.z = 0.0;
    }

    return(n);
}
```

Funktionen, die nur innerhalb von `vector3d.c` verwendet werden und nicht von außen sichtbar sein sollen, müssen als `static` definiert werden. Ihre Prototypen dürfen auch nicht in der `.h`-Datei stehen, sondern sollten am Anfang der `.c`-Datei stehen.

Der Datentyp könnte dann z. B. so verwendet werden:

```
#include "vector3d.h"

...

Vector3d a;
Vector3d n;

a = v3d_new(10.0, 5.0, -3.0);
n = v3d_normalized(a);
```

5.2 Referenzierte Datentypen in C

Für große `struct`s ist die Vorgehensweise im vorherigen Abschnitt sehr ineffektiv, denn die Übergabe und Rückgabe von großen Datenstrukturen ist mit dem Kopieren von viel Speicherplatz verbunden. Alternativ können wir ausschließlich mit Adressen (also Referenzen) arbeiten.

Da wir nur noch Referenzen speichern, deklarieren wir nur einen Typ `rVector3d` als Adresse auf unsere `struct` (nicht mehr die `struct` selbst). Eine Variable vom Typ `rVector3d` ist keine `struct`-Variable, sondern ein Zeiger darauf. `struct`-Variablen verwenden wir nicht mehr, deswegen müssen wir uns auch den Speicherplatz selbst mit `malloc()` reservieren, d. h. unser Konstruktor ruft jetzt `malloc()` auf. Entsprechend brauchen wir auch einen *Destruktor*, der `free()` aufruft. Außerdem lohnt sich gelegentlich ein sogenannter *Copy-Constructor*, der eine Kopie einer existierenden Variable dieses Typs herstellt.

Um nicht in Namenskonflikte zu geraten, nehmen wir als neues Kürzel für Funktionen `rv3d...` und nennen die `.c`-Datei für den Typ `rVector3d` entsprechend `rvector3d.c`. Die Datei `rvector3d.h` könnte so aussehen:

```
#if !defined RVECTOR3D_H
#define RVECTOR3D_H

#include <stdlib.h>
#include <math.h>

typedef struct
{
   double x;
   double y;
   double z;
} *rVector3d;

#define rv3d_get_x(v) ((v)->x)
#define rv3d_get_y(v) ((v)->y)
#define rv3d_get_z(v) ((v)->z)

#define rv3d_set_x(v, new_x) \
   ((v)->x = (new_x))
#define rv3d_set_y(v, new_y) \
   ((v)->y = (new_x))
#define rv3d_set_z(v, new_z) \
   ((v)->z = (new_x))

rVector3d rv3d_new(double x,
   double y, double z);
rVector3d v3d_copy(rVector3d v);
void rv3d_delete(rVector3d v);

double rv3d_get_length(
   rVector3d v);
rVector3d rv3d_normalized(
   rVector3d v);

#endif
```

In der entsprechenden Datei `rvector3d.c` müssen wir jetzt leicht umdenken, weil `rVector3d` ein Adresstyp ist:

```
#include "rvector3d.h"

rVector3d rv3d_new(double x,
   double y, double z)
{
```

```
   rVector3d v;

   v = (rVector3d)malloc(
      sizeof(*v));
   if (NULL == v)
   {
      return(NULL);
   }
   v->x = x;
   v->y = y;
   v->z = z;

   return(v);
}

rVector3d rv3d_copy(rVector3d v)
{
   rVector3d new;

   new = rv3d_new(v->x, v->y,
      v->z);

   return(new);
}

void rv3d_delete(rVector3d v)
{
   free((void *)v);
}

double rv3d_get_length(rVector3d v)
{
   return(sqrt(v->x * v->x +
      v->y * v->y + v->z * v->z));
}

rVector3d rv3d_normalized(
   rVector3d v)
{
   rVector3d n;
   double length;

   length = rv3d_get_length(v);

   if (length > 0.0)
   {
      n = rv3d_new(v->x / length,
         v->y / length,
         v->z / length);
   }
   else
   {
      n = rv3d_new(0.0, 0.0, 0.0);
   }
   return(n);
}
```

Das Anwendungsbeispiel sieht dann so aus:

```
#include "rvector3d.h"
...
rVector3d a;
rVector3d n;

a = rv3d_new(10.0, 5.0, -3.0);
if (NULL != a)
{
   n = rv3d_normalized(a);
   if (NULL != n)
   {
      ...
   }
   rv3d_delete(n);
}
rv3d_delete(a);
```

Den Destruktor `rv3d_delete()` immer aufrufen zu müssen, ist natürlich lästig. Noch schlimmer sind aber die vielen Aufrufe von `malloc()` (beziehungsweise des Konstruktors) und die damit verbundenen Fehlerabfragen. Deswegen werden bei referenzierten Datentypen eher Prozeduren geschrieben, die Variablen ändern, anstatt neue zu erzeugen. Also z. B. eine Prozedur `normalize()`, die einen Vektor normalisiert, anstatt einer Funktion `normalized()`, die einen normalisierten Vektor zurückgibt. Beispiel:

```
void rv3d_normalize(
   rVector3d v)
{
   double length;

   length = rv3d_get_length(v);

   if (length > 0.0)
   {
      v->x = v->x / length,
      v->y = v->y / length,
      v->z = v->z / length,
   }
}
```

Damit würde die Normalisierung von vorhin so aussehen:

```
#include "rvector3d.h"
...
rVector3d a;

a = rv3d_new(10.0, 5.0, -3.0);
if (NULL != a)
{
   rv3d_normalize(a);
   ...
}
rv3d_delete(a);
```

Das ist schon deutlich weniger aufwendig als das die Anwendung von `rv3d_normalized`.

Kapitel 6

Zeichnen geometrischer Objekte mit OpenGL

(Dieses Kapitel ist mehr oder weniger eine Übersetzung des zweiten Kapitels des „OpenGL Programming Guide, Third Edition“ von Woo, Neider, Davis und Shreiner.)

Obwohl mit OpenGL sehr komplexe und interessante Bilder gezeichnet werden, sind sie doch alle aus einer kleinen Zahl primitiver graphischer Elemente aufgebaut. Auf der höchsten Abstraktionsstufe gibt es drei grundlegende Operationen zum Zeichnen: Löschen des Fensters, Zeichnen eines geometrischen Objekts und Zeichnen eines Rasterbilds; letzteres wird hier nicht behandelt. Vielmehr beschränken wir uns auf das Zeichnen geometrischer Objekte, wie Punkte, gerade Striche und flache Polygone.

Aus diesen wenigen Teilen können schon beeindruckende Szenarien entstehen, wenn nur genug einfache Objekte zusammengesetzt werden, um z. B. sanft gebogene Flächen oder Kurven darzustellen. Daneben erlaubt OpenGL eine Menge von speziellen Algorithmen, die es zum Beispiel Spielen ermöglichen immer realistischere Bilder auch in Echtzeit zu erzeugen, obwohl in diesen Fällen verhältnismäßig wenige Polygone gezeichnet werden. Auf diese Tricks wird hier nicht eingegangen. (Sie werden zum Teil in der Vorlesung „Bildsynthese“ im Sommersemester von Prof. Thomas Ertl besprochen.) Vielmehr soll hier die grundlegende Vorgehensweise beim Zeichnen einfacher Objekte mit OpenGL erklärt werden.

6.1 Ein Survival-Kit zum Zeichnen

In diesem Abschnitt geht es noch nicht um geometrische Objekte, sondern um Voraussetzungen, wie einem gelöschten Fenster, der Auswahl einer Farbe, dem Leeren den OpenGL-Kommandobuffers, um die Ausgabe zu erzwingen, und die Spezifikation einfacher Koordinatensysteme.

6.1.1 Löschen des Fensters

Vor dem Zeichnen in ein Fenster sollte zunächst der Inhalt gelöscht werden, d. h. alle Bildpunkte auf eine gemeinsame Hintergrundfarbe gesetzt werden. Natürlich könnte das durch das Zeichnen eines großen Rechtecks erreicht werden, aber ein spezieller Befehl zum Löschen des Fensters erlaubt eine schnellere Implementierung — vor allem in Hardware.

Der Befehl zum Setzen dieser Hintergrundfarbe wurde schon erwähnt. Hier die Funktionsdeklaration:

```
void glClearColor(GLclampf Rot,
GLclampf Grün, GLclampf Blau,
GLclampf Alpha);
```

Damit werden die *Rot*-, *Grün*-, *Blau*- und *Alpha*-Anteile der Hintergrundfarbe gesetzt, die alle auf das Intervall $[0, 1]$ ge*clamp*t werden, was durch den Datentyp `GLclampf` deutlich gemacht wird. *Clamping* heißt in diesem Fall, dass Werte kleiner 0 auf 0 und Werte größer 1 auf 1 gesetzt werden.

Für OpenGL hat jedes Pixel nicht nur eine Farbe, sondern auch andere Eigenschaften, wie eine *Tiefe* (bei dreidimensionalen Graphiken), eventuell eine

Maske, die ihn gewissermaßen unsichtbar für bestimmte Zeichenoperationen macht, oder auch weitere Farben, die unabhängig von der dargestellten Farbe gespeichert werden. Diese Informationen sind in verschiedenen Buffern gespeichert.

Der Befehl zum Löschen der Buffer eines Fensters ist

```
void glClear(GLbitfield Bitmaske);
```

Die *Bitmaske* ergibt sich aus der bitweisen VerODERung der Konstanten `GL_COLOR_BUFFER_BIT` (für den Farbbuffer), `GL_DEPTH_BUFFER_BIT` (für den Tiefen- oder z-Buffer), `GL_ACCUM_BUFFER_BIT` (für den Accumulation-Buffer) und `GL_STENCIL_BUFFER_BIT` (für den Stencil-Buffer).

Also kann mit

```
glClear(GL_COLOR_BUFFER_BIT |
   GL_DEPTH_BUFFER_BIT);
```

sowohl der Farb- als auch z-Buffer gelöscht werden, was eventuell schneller ist, als die beiden Buffer separat zu löschen. Löschen ist tatsächlich eine verhältnismäßig aufwendige Operation, da jeder Pixel betroffen ist.

6.1.2 Spezifizieren einer Farbe

Farbberechnungen können in OpenGL durch Beleuchtungen und Texturen sehr aufwendig sein. Im einfachsten Fall wird aber lediglich eine Farbe spezifiziert, die dann für alle Objekte verwendet wird, bis eine neue Farbe angegeben wird. Diese Vorgehensweise ist wesentlich effizienter als für jedes Objekt eine Farbe angeben zu müssen, da oft viele Objekte die gleiche Farbe besitzen.

Eine Farbe wird mit `glColor3f(`*Rot*, *Grün*, *Blau*`)` angegeben. Man kann sich vorstellen, dass die drei Anteile (additiv) gemischt werden. 0.0 bedeutet, dass von einem Anteil nichts genommen wird; 1.0 bedeutet, dass möglichst viel davon genommen wird. Damit ergeben sich z. B. diese Farben:

Farbe	*Rot*	*Grün*	*Blau*
Schwarz	0.0	0.0	0.0
Rot	1.0	0.0	0.0
Grün	0.0	1.0	0.0
Gelb	1.0	1.0	0.0
Blau	0.0	0.0	1.0
Violett	1.0	0.0	1.0
Türkis	0.0	1.0	1.0
Weiß	1.0	1.0	1.0

OpenGL kennt noch eine vierte Komponente (Alpha), die von `glColor3f` auf 0.0 gesetzt wird. Der Alpha-Anteil kann z. B. genutzt werden, um transparente Farben zu spezifizieren.

6.1.3 Leeren des OpenGL-Kommandobuffers

OpenGL-Kommandos werden in einer Pipeline bearbeitet, womit schon ein gewisser Grad an Parallelisierung erreicht wird. In einigen Fällen werden Kommandos aber auch gebuffert, um sie später effizienter übertragen oder ausführen zu können. Mit

```
void glFlush(void);
```

kann sichergestellt werden, dass zumindest angefangen wird, alle Kommandos tatsächlich auszuführen.

Falls das nicht ausreicht, kann mit

```
void glFinish(void);
```

solange gewartet werden, bis alle OpenGL-Kommandos beendet wurden.

6.1.4 Einfache Koordinatensysteme

Die Angabe einer dreidimensionalen Kameraposition und -richtung ist naturgemäß etwas kompliziert, insbesondere wenn die Kamera (wie jede gute Kamera) zentralperspektivische Bilder machen soll. OpenGL kann zentralperspektivische Darstellungen berechnen, aber hier soll nur ein ganz einfaches parallelperspektivisches Koordinatensystem spezifiziert werden.

```
glViewport(0, 0, (GLsizei)b,
   (GLsizei)h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho(xmin, xmax,
   ymin, ymax);
```

spezifiziert ein Koordinatensystem in einem Fenster der Pixelhöhe `h` und der Pixelbreite `b` mit dem Punkt (`xmin`,`ymin`) in der linken unteren Ecke des Fensters und (`xmax`,`ymax`) in der rechten, oberen Ecke.

6.2 Spezifizieren von Punkten, Linien und Polygonen

Die geometrischen Primitive von OpenGL haben nicht ganz die Bedeutung der gleichnamigen mathematischen

Objekte. Positionen können z. B. nicht beliebig genau angegeben werden und auf realen Bildschirmen können die Objekte nicht mit beliebig hoher Auflösung wiedergegeben werden.

6.2.1 Punkte

Ein Punkt wird mit OpenGL durch einen Vertex angegeben, wobei die Koordinaten des Vertex durch drei Gleitkommazahlen spezifiziert werden. Vertizes sind für OpenGL immer dreidimensional, falls keine dritte Koordinate (z) angegeben wurde, wird 0.0 angenommen.

6.2.2 Linien

... sind eigentlich Linienstücke, also die gerade Verbindung zwischen den Vertizes an den Endpunkten. Durch Verbindung von vielen Linienstücken an den jeweiligen Endpunkten können auch „glatte“ Kurven dargestellt werden. Zumindest so glatt, dass die Zusammensetzung aus geraden Stücken nicht mehr auffällt.

6.2.3 Polygone

... bestehen aus der Fläche, die von einer Folge von Linienstücken eingeschlossen wird. Die Linienstücke werden durch die Vertizes an ihre Endpunkte, also den Ecken des Polygons, beschrieben.

OpenGL garantiert nicht, dass jedes Polygon richtig dargestellt wird, denn das wäre sehr schwer in Hardware zu implementieren. Aber OpenGL kann jedes *flache, konvexe und einfache Polygon* darstellen. Ein *konvexes* Polygon ist dadurch gekennzeichnet, dass die Verbindungslinie zwischen allen Paaren von Punkten innerhalb des Polygons wieder im Polygon liegt. Ein *einfaches* Polygon ist ein Polygon ohne Löcher oder Selbstüberschneidungen der Kanten. Bei einem *flachen* Polygon müssen alle Eckpunkte auf einer Ebene liegen. Wenn alle drei Eigenschaften erfüllt sind, ist sicher, dass OpenGL das Polygon zeichnen kann. Das klingt nach reichlich vielen Einschränkungen, andererseits stellt sich heraus, dass *jedes* Dreieck *flach*, *konvex* und in dem oberen Sinn *einfach* ist.

Um andere Polygone zeichnen zu können müssen sie zuerst in Polygone aufgeteilt werden, die diese Forderungen erfüllen. Dazu gibt es Funktionen in der GLU-Bibliothek, die hier aber nicht besprochen werden.

6.2.4 Spezifizieren von Vertizes

OpenGL beschreibt jedes geometrische Objekt letztlich als eine Folge von Vertizes, die mit `glVertex*()` angegeben werden:

`void glVertex{234}{sifd}[v](`*Typ Koordinaten*`);`

Vertizes in OpenGL sind zwar dreidimensional, können aber auch mit vier *homogenen* Koordinaten angegeben werden, so dass bis zu vier Koordinaten möglich sind. Bei der Variante mit zwei Koordinaten wird die dritte Koordinate auf 0.0 gesetzt. Für den Datentyp sind `GLshort` (`s`), `GLint` (`i`), `GLfloat` (`f`) und `GLdouble` (`d`) möglich. Die Vektor-Variante (durch ein `v` markiert) kann eventuell schneller sein. Beispiele:

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.14159);
glVertex4f(2.3, 1.0, -2.2, 2.0);
{
   GLdouble dfeld[3] = {5., \
      9., 1.};
   glVertex3dv(dfeld);
}
```

Das erste Beispiel spezifiziert einen dreidimensionalen Vertex mit Koordinaten $(2, 3, 0)$. (Die dritte Koordinate wird automatisch auf 0 gesetzt.) Im zweiten Beispiel werden alle drei Koordinaten als doppelt genaue Gleitkommazahlen angegeben. Das dritte Beispiel benutzt vier homogene Koordinaten und spezifiziert deshalb den Vertex $(2.3/2.0, 1.0/2.0, -2.2/2.0)$. (Homogene Koordinaten werden durch eine Division der ersten drei Koordinaten durch die vierte Koordinate in gewöhnliche dreidimensionale Koordinaten verwandelt.)

6.2.5 OpenGL Primitive

Aus einer Liste von Vertizes kann OpenGL verschiedene geometrische Primitive erzeugen. Dazu müssen die Vertizes zwischen `glBegin()` und `glEnd()` angegeben werden. Das Argument von `glBegin()` gibt dabei an, welche geometrischen Objekt aus den Vertizes erzeugt werden sollen.

`void glBegin(GLenum` *Modus*`);`

markiert den Start einer Reihe von Vertizes, die geometrische Objekte beschreiben. *Modus* gibt die Art der geometrischen Objekte an. Tabelle 6.1 enthält die erlaubten Konstanten und deren Bedeutung.

Konstante	Bedeutung
GL_POINTS	einzelne Punkte
GL_LINES	getrennte Linienstücke
GL_LINE_STRIP	zusammenhängende Linienstücke
GL_LINE_LOOP	wie GL_LINE_STRIP aber geschlossen
GL_TRIANGLES	getrennte Dreiecke
GL_TRIANGLE_STRIP	Band aus Dreiecken
GL_TRIANGLE_FAN	Fächer aus Dreiecken
GL_QUADS	getrennte Vierecke
GL_QUAD_STRIP	Band aus Vierecken
GL_POLYGON	einfaches, convexes und flaches Polygon

Tabelle 6.1: Namen und Bedeutungen von geometrischen Primitiven.

Funktion	setzt...
glVertex*()	Vertex-Koordinaten
glColor*()	aktuelle Farbe
glNormal*()	aktuelle Normale
glTexCoord*()	aktuelle Texturkoord.
glEdgeFlag*()	aktuellen Kantenstil

Tabelle 6.2: Funktionen zum Setzen von Vertex-spezifischen Informationen.

Wie die Vertizes genau interpretiert werden, um die Primitive zu erzeugen, soll hier nicht erklärt werden. Offensichtlich gibt es verschiedene Möglichkeiten Linienstücke, Dreiecke oder Vierecke zu spezifizieren. Welche die beste ist, hängt von den Vertexdaten ab.

Zwischen `glBegin()` und `glEnd()` werden aber nicht nur Vertizes spezifiziert, sondern auch Eigenschaften der Vertizes, wie Farben, Normalen, Texturkoordinaten, etc. In Tabelle 6.2 sind ein paar von den dazu verwendeten Befehlen aufgelistet. Wie mit `glColor*()` immer eine aktuelle Farbe gesetzt wird, die dann für Vertizes verwendet wird, werden auch mit den anderen Funktionen aktuelle Werte gesetzt. Nur `glVertex*()` erzeugt einen Vertex und verwendet dazu die jeweils aktuellen Werte.

Zwischen `glBegin()` und `glEnd()` sind nicht viele andere OpenGL-Kommandos erlaubt, aber natürlich alle C-Konstruktionen.

6.3 Darstellen von Punkten, Linien und Polygonen

Normalerweise werden Punkte als einzelne Pixel gezeichnet, Linien sind auch einen Pixel dick und durchgezogen und Poygone werden gefüllt. Diese Voreinstellungen können aber auch geändert werden:

```
void glPointSize(GLfloat Breite);
```

setzt die Punktbreite in Punkten. Und

```
void glLineWidth(GLfloat Breite);
```

setzt entsprechend die Linienbreite in Punkten. Mit

```
void glLineStipple(GLintFaktor,
GLushortBitmuster);
```

kann angeben werden, wie Linien gestrichelt werden sollen. Nach einem Aufruf von `glLineStipple()` muss noch mit `glEnable(GL_LINE_STIPPLE)` das Stricheln von Linien aktiviert werden. (Mit `glDisable(GL_LINE_STIPPLE)` wird es wieder deaktiviert.) *Bitmuster* wird als Binärzahl aufgefasst und gibt so ein Muster an, das zum Stricheln verwendet wird. Die Skalierung dieses Musters kann mit *Faktor* angegeben werden.

Für Polygone unterscheidet OpenGL zwischen der Vorder- und der Rückseite eines Polygones. Entsprechend können Vorder- und Rückseite verschieden dargestellt werden. Wie OpenGL festlegt, was die Vorderseite ist, wird mit

```
glFrontFace(GLenum Modus);
```

festgelegt. *Modus* kann entweder `GL_CCW` (counterclockwise: gegen den Uhrzeigersinn) oder `GL_CW` (clockwise: im Uhrzeigersinn) sein. Normalerweise gilt `GL_CCW`, d. h. von Polygone, deren Vertizes auf dem Bildschirm entgegen dem Uhrzeigersinn erscheinen, ist die Vorderseite zu sehen. Falls die Vertizes auf dem Bildschirm im Uhrzeigersinn erscheinen, ist die Rückseite zu sehen. Mit `GL_CW` dreht sich diese Definition der Vorder- und Rückseite gerade um.

Für Vorder- und Rückseite kann mit

```
void glPolygonMode(GLenum Seite, GLenum Modus);
```

getrennt angegeben werden, ob nur die Eckpunkte (*Modus* = `GL_POINT`), nur die Kanten (*Modus* = `GL_LINE`) oder die Polygonfläche (*Modus* = `GL_FILL`) dargestellt werden sollen. *Seite* gibt dabei die betroffene Seite an, und kann `GL_FRONT`, `GL_BACK` oder `GL_FRONT_AND_BACK` sein.

Falls nur die Polygonkanten dargestellt werden, spielen die *edge flags* der Vertizes eine Rolle. Mit

```
void glEdgeFlag(GLboolean flag);
```

kann für jeden Vertex ein *edge flag* angegeben werden. Das *edge flag* entscheidet, ob eine Polygonkante, die an diesem Vertex beginnt, gezeichnet werden soll.

Unter Umständen ist es sinnvoll, die Vorder- oder Rückseite gar nicht darzustellen (sogenanntes *Face-Culling*). Das muss mit `glEnable(GL_CULL_FACE)` aktiviert werden. Ob Vorder- oder Rückseite „gecullt“ werden, kann mit

```
void glCullFace(GLenum Seite);
```

eingestellt werden. (*Seite* kann wieder die gleichen Werte wie oben annehmen.)

Das „Stricheln“ von Polygonen wird hier nicht erklärt, vor allem weil der gleiche Effekt mit Texturen viel besser erreicht werden kann.

6.4 Normalen

Mit OpenGL kann an jedem Vertex eine Normalenrichtung definiert werden, also eine Richtung die senkrecht auf einer Fläche steht. Die Funktion heißt

```
void glNormal3f(nx, ny, nz)
```

wobei *(nx, ny, nz)* den Normalenvektor bezeichnet. Diese Normalenrichtung wird vor allem zur Berechnung der Beleuchtung verwendet. In zwei Dimensionen machen Beleuchtung und Normalen aber wenig Sinn, deswegen werden sie hier nicht weiter erklärt.

Übungen zum dritten Tag

Verwenden Sie ab jetzt immer Makefiles für die Übersetzung der Programme!

Aufgabe III.1

Schreiben Sie eine Funktion, die zwei Punkte $\begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$ und $\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$ verbindet. Die Funktion soll mit OpenGL einen geraden Strich zeichnen, wenn das Quadrat des Abstands der Punkte $(x_0 - x_1)^2 + (y_0 - y_1)^2$ kleiner 4 Bildpunkte ist. Sonst soll sich die Funktion selbst aufrufen, um folgende Punktepaare zu verbinden:$(\vec{a}, \vec{b})$, $(\vec{b}, \vec{c})$,$(\vec{c}, \vec{d})$ und $(\vec{d}, \vec{e})$. Mit

$$\begin{aligned}
\vec{a} &= \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}, \\
\vec{b} &= \begin{pmatrix} (2x_0 + x_1)/3 \\ (2y_0 + y1)/3 \end{pmatrix}, \\
\vec{c} &= \begin{pmatrix} (x_0 + x_1)/2 - \sqrt{3}/6(y_1 - y_0) \\ (y_0 + y_1)/2 + \sqrt{3}/6(x_1 - x_0) \end{pmatrix}, \\
\vec{d} &= \begin{pmatrix} (x_0 + 2x_1)/3 \\ (y_0 + 2y1)/3 \end{pmatrix}, \\
\vec{e} &= \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}.
\end{aligned}$$

Testen Sie Ihre Funktion, indem Sie die Eckpunkte eines Dreiecks verbinden. Für ein gleichseitiges Dreieck sollte die sogenannte Kochsche Schneeflockenkurve entstehen.

Eine Funktion, die sich selbst aufruft, wird rekursiv genannt. Dadurch entsteht eine Schachtelung von Aufrufen. Die Tiefe dieser Schachtelung wird Rekursionstiefe genannt. Ändern Sie das Programm so, dass auch dann eine gerade Linie gezeichnet wird, wenn die Rekursionstiefe einen einstellbaren Wert erreicht hat.

Aufgabe III.2

Turtle-Graphics (Schildkröten-Graphiken) sind ein wesentlicher Bestandteil der Programmiersprache Logo. Die Idee dabei ist, mit Hilfe von einfachen Kommandos eine Schildkröte, die eine Spur hinterlässt, auf dem Bildschirm zu bewegen. Geht die Schildkröte z. B. 10 Bildpunkte vorwärts, dann entsteht eine Linie, die 10 Punkte lang ist. Die Kommandos zum Bewegen der Schildkröte sind: Setzen der Schildkröte an einem angegebenen Bildpunkt mit einer angegeben Blickrichtung, Vorwärtsgehen um eine angegebene Länge (rückwärts bei negativer Länge) und Drehen nach rechts um einen angegebenen Winkel (links bei negativem Winkel). Entsprechend wird der Zustand der Schildkröte vollständig durch ihre Position und einen Winkel spezifiziert, der ihre Blickrichtung angibt.

Implementieren Sie diese drei Kommandos mit Hilfe von OpenGL. Achten Sie auf eine saubere Trennung von Deklaration und Implementierung der Funktionen. Schreiben Sie ein Testprogramm, das mit Hilfe Ihrer Schildkröte eine gleichseitiges Sechseck zeichnet.
(Bei Interesse, können Sie auch versuchen, die Kochsche Schneeflockenkurve aus Aufgabe III.1 mit Hilfe der Schildkröten-Kommandos zu Zeichen.)

Aufgabe III.3

In dieser und in den folgenden Aufgaben soll ein ganz einfacher Raytracer implementiert werden. Also ein Programm, das mittels *ray-tracing* (Sehstrahlenverfolgung) das Bild einer dreidimensionalen Szene darstellt.

Ein Raytracer benutzt sehr viel Vektoralgebra, deshalb soll zuerst eine einfache Bibliothek von Vektorfunktionen implementiert werden. Verwenden Sie dazu den in Kapitel 5 vorgestellten Vektordatentyp und erweitern diesen um folgende Opertationen:

- einen mit einem Skalar s multiplizierten Vektor $s\vec{a}$ berechnen,
- die Summe $\vec{a} + \vec{b}$ von zwei Vektoren $\vec{a}$ und $\vec{b}$ berechnen,
- den Differenzvektor $\vec{a} - \vec{b}$ von zwei Vektoren $\vec{a}$ und $\vec{b}$ berechnen ($\vec{a} - \vec{b}$ ist der Vektor von $\vec{b}$ nach $\vec{a}$),
- das Skalarprodukt zweier Vektoren $\vec{a}$ und $\vec{b}$ berechnen: $\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$,
- das Kreuzprodukt zweier Vektoren $\vec{a}$ und $\vec{b}$ berechnen: $\vec{a} \times \vec{b} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$,
- die Projektion eines Vektors $\vec{a}$ auf einen Vektor $\vec{b}$ berechnen: $\vec{a}_{\vec{b}} = \vec{b}\frac{\vec{a} \cdot \vec{b}}{|\vec{b}|^2}$,
- die Reflektion $\vec{a}'$eines Vektors $\vec{a}$ an einer Ebene berechnen, die durch einen Normalenvektor $\vec{b}$ gegeben ist: $\vec{a}' = \vec{a} - 2\vec{a}_{\vec{b}}$.

Trennen Sie die Deklaration dieser Funktionen von der eigentlichen Implementierung und schreiben Sie ein einfaches Testprogramm für ihre Funktionen.

Aufgabe III.4

Schreiben Sie einen Raytracer, der eine Szene aus mehreren Kugeln zeichnet. Ein Raytracer schickt von einem virtuellen Augpunkt (in einem virtuellen Raum) je einen Sehstrahl zu jedem Bildpunkt einer virtuellen Bildebene, die genauso viele Bildpunkte hat, wie das Bild (Fenster), das berechnet wird. Für jeden dieser Sehstrahlen wird dann eine Farbe berechnet, die in dem Bild (Fenster) gesetzt wird.

Um die Sache zu vereinfachen, geben Sie der virtuellen Bildebene die gleichen Koordinaten (z-Koordinate gleich 0), wie dem OpenGL-Fenster. Die positive z-Achse geht in OpenGL üblicherweise senkrecht aus der Bildebene heraus auf den Betrachter zu. Wählen Sie in diesem dreidimensionalen Koordinatensystem einen geeigneten Augpunkt. Berechnen Sie dann normierte Richtungsvektoren vom Augpunkt zu jedem Bildpunkt des OpenGL-Fensters.

Fügen Sie nun Kugeln mit unterschiedlichem Durchmesser und Position in Ihre Szene ein. Schneiden Sie dazu jeden Sehstrahl mit allen Kugeln und finden Sie denjenigen Schnittpunkt, der am nächsten zum Betrachter ist. (Außerdem muss der Schnittpunkt natürlich *vor* (in Blickrichtung) und nicht *hinter* dem Betrachter liegen.) Färben Sie dann den entsprechenden Bildpunkt anhand der Koordinaten dieses Schnittpunktes ein.

Hinweis: Die beiden Schnittpunkte $\vec{s}_{1,2}$ eines Strahls (von Punkt $\vec{a}$ in Richtung $\vec{b}$) mit einer Kugel (mit Radius r und Mittelpunkt $\vec{m}$) sind gegeben durch $\vec{s}_{1,2} = \vec{a} + \lambda_{1,2}\vec{b}$. Mit

$$\lambda_{1,2} = \frac{1}{\vec{b}^2}\Big(-\vec{b}\cdot(\vec{a}-\vec{m}) \pm \pm\sqrt{(\vec{b}\cdot(\vec{a}-\vec{m}))^2 - \vec{b}^2((\vec{a}-\vec{m})^2 - r^2)}\Big)$$

(Falls der Ausdruck unter der Wurzel negativ werden sollte, gibt es keine Schnittpunkte.)

Zusatzaufgabe III.5

Durch Bearbeiten dieser Aufgabe können Zusatzpunkte erreicht werden. Ihr Übungsbetreuer kann Ihnen nähere Information zur Abgabe geben.

Erweitern Sie den Raytracer um eine oder mehrere der folgenden Funktionalitäten.

Hintergrund

Die Kugeln schweben bisher noch im leeren Raum. Schöner wäre es aber, wenn im Hintergrund z.B. ein Wüste unter blauem Himmel zu sehen wäre. Färben Sie dazu jedes Pixel, dessen Sehstrahl keine Ihrer Kugeln trifft, entsprechend der folgenden Vorschrift ein:
Die Farbe für jeden Sehstrahl sei *Rot-Anteil* = *Grün-Anteil* = $\frac{1}{2} - \operatorname{atan}(100y)/\pi$ und *Blau-Anteil* = $\frac{1}{2} + \operatorname{atan}(100y)/\pi$, wobei y die zweite Komponente der normierten Richtung des Sehstrahls ist.

Beleuchtung

Um den Kugeln einen besseren räumlichen Eindruck zu geben, müssen deren Oberflächen beleuchtet werden. Ein sehr einfaches und in der Computergraphik häufig eingesztes Beleuchtungsverfahren ist das Phong-Modell. Eine gute Beschreibung dieses Beleuchtungsmodells finden Sie bspw. bei *Wikipedia*.

Färben Sie die Punkte, auf den Kugeloberflächen entsprechend des Phong-Modells ein. Untersuchen Sie die Effekte, die Sie durch unterschiedliche Einstellung der Lichtfarbe, der Lichtposition und der verschiedenen Konstanten erreichen können.

Reflektion

Mit Hilfe von Raytracing können auch auf einfache Weise Reflektionen berechnet werden. Dazu muss, nachdem der nächstgelegene Schnittpunkt mit einer Kugel berechnet wurde, ein neuer Sehstrahl von diesem Schnittpunkt aus, in Richtung des reflektierten Sehstrahls verfolgt werden. Zur Berechnung dieser Reflektion ist die Normalenrichtung $\vec{n}$ der Kugeloberfläche nötig. Die ist aber einfach $\vec{n} = (\vec{s} - \vec{m})/r$, wobei $\vec{s}$ der Schnittpunkt ist, $\vec{m}$ der Mittelpunkt der geschnittenen Kugel und r ihr Radius. Wird der Richtungsvektor des Sehstrahls an $\vec{n}$ reflektiert (s. Aufgabe III.3), dann kann vom Schnittpunkt aus mit der reflektierten Richtung ein neuer Sehstrahl gestartet werden. D. h. auch dieser Sehstrahl muss wieder mit allen Kugeln geschnitten werden, um den nächstgelegenen Schnittpunkt zu finden, an dem der Sehstrahl wieder reflektiert wird, etc. (Vermeiden Sie die Implementierung einer Endlosrekursion!)

Teil IV

Vierter Tag

Kapitel 7

Standardbibliotheken

7.1 Die Standardbibliotheken

C ist eine verhältnismäßig „kleine“ Sprache, zum Beispiel gemessen an der Zahl der Keywords. Das ist deswegen möglich, weil viel Funktionalität nicht in die Programmiersprache aufgenommen wurde, sondern in standardisierte Funktionsbibliotheken zu finden ist. Zum Beispiel besitzt die Programmiersprache C keine Kommandos zur Ein-/Ausgabe, aber mit Hilfe von `printf()` aus der Standardbibliothek können trotzdem Textausgaben produziert werden.

Dieses Design von C wurde gelegentlich kritisiert, aber es hat sich bezahlt gemacht. Zum Beispiel hat sich die Ein-/Ausgabe von Computerprogrammen seit den 70er Jahren so stark geändert, dass `printf()` in Programmen mit graphischer Oberfläche praktisch nicht mehr verwendet wird. Das ist nicht allzu schlimm, weil es nur heißt, dass die Funktion `printf()` aus einer Standardbibliothek weniger genutzt wird. Wäre ein derartiger Ausgabemechanismus aber direkt Teil der Sprache C, dann würde dieser Teil von C inzwischen veraltet sein und müsste überarbeitet werden. Änderungen in einer Computersprache durchzuführen ist aber sehr schwierig, weil es unter Umständen dazu führt, dass Programme geändert werden müssen.

Es gibt sehr viele Funktionen in der Standardbibliothek. Einige wie `printf()` (in `stdio.h` deklariert), `malloc()` und `free()` (`malloc.h` und `stdlib.h`) und ein paar mathematische Funktionen (`math.h`) wurden schon erwähnt. Wichtig sind auch die Dateioperationen aus `stdio.h`. Dabei tauchen oft Fehler auf, die mit Hilfe der Deklarationen in `errno.h` diagnostiziert werden können. Um Strings (Zeichenketten) zu verarbeiten, sind die Funktionen aus `string.h` unentbehrlich.

Tabelle 7.1 listet ein paar wenige nützliche `.h`-Dateien aus der Standardbibliothek auf, es gibt noch viel mehr. Und noch wesentlich mehr Funktionen. Um damit zurecht zu kommen, noch ein paar Tips:

- Der C-Compiler muss `.h`-Dateien lesen können, deshalb kann der Programmierer sie normalerweise auch lesen (unter UNIX sind sie meistens unter `/usr/include` gespeichert). `.h`-Dateien zu lesen lohnt sich aber nicht nur bei Standard-, sondern bei allen Bibliotheken, denn dort sind zumindest die verfügbaren Funktionen deklariert, wenn nicht sogar ihre Funktion kommentiert.
- Unter UNIX sind zu den meisten Funktionen aus der Standardbibliothek *man-pages* verfügbar. Mit `man printf` werden z. B. die Details der Formatangaben erklärt.
- Viele C-Entwicklungsumgebungen enhalten auch Erklärungen zu den Bibliotheksfunktionen. Darin sind normalerweise auch Beschreibungen der `.h`-Dateien und Verweise auf andere Funktionen, so dass die meisten Funktionen relativ schnell gefunden werden können, auch wenn ihr Name unbekannt ist.
- Es sollte klar geworden sein, dass die jeweilige `.h`-Datei `#included` werden sollte. Im Prinzip ist dies nicht unbedingt nötig, da die Deklarationen auch direkt in das Programm geschrieben werden können. Davon ist aber dringend abzuraten.

Im Folgenden werden noch einige nützliche Funktionen aus der Standardbibliothek vorgestellt, die im Leben eines C-Programmierers häufiger vorkommen.

7.1.1 Dateioperationen

Als einfaches Beispiel für die Benutzung der Dateioperationen aus `stdio.h` sollen hier die Zeilen aus einer Textdatei namens `main.c` ausgelesen werden:

.h-Datei	Beschreibung
errno.h	Fehlerabfragen
malloc.h	Speicherverwaltung
math.h	mathematische Funktionen
stdio.h	Ein-/Ausgabe
stddef.h	Standarddefinitionen
stdlib.h	nützliche Deklarationen
string.h	String-Verarbeitung
ctype.h	Zeichenoperationen
time.h	Zeitabfragen
assert.h	Debugoperationen

Tabelle 7.1: Beispiele von .h-Dateien der Standardbibliothek.

```
#include <stdio.h>

#define MAX_LAENGE 100

int main(void)
{
   FILE *datei;
   char textzeile[MAX_LAENGE];

   datei = fopen("main.c", "r");
   if (NULL == datei)
   {
      return(1);
   }
   while (!feof(datei))
   {
      fgets(textzeile, MAX_LAENGE,
         datei);

      printf("%s", textzeile);
   }
   fclose(datei);
   return(0);
}
```

Mit

```
datei = fopen("main.c", "r");
```

wird die Adresse auf eine FILE-Struktur zurückgegeben, falls fopen() die Datei namens main.c erfolgreich zum Lesen ("r" für *read*) öffnen konnte. (Sonst wird NULL zurückgegeben.)

In der folgenden while-Schleife wird dann die Datei Zeile für Zeile gelesen. Das Ende dieser Schleife ist erreicht, wenn das Ende der Datei (end of file = eof) erreicht wurde, was sich bequem mit feof() abfragen lässt. (feof() gibt genau dann einen Wert ungleich 0 zurück, wenn das Dateiende erreicht wurde.)

fgets(textzeile, MAX_LAENGE, datei) liest aus der Datei bis zu 99 Zeichen ein und speichert sie in textzeile. (fgets() steht für "file get string".) fgets stoppt mit dem Einlesen bei jedem Zeilenumbruch oder wenn die Datei zu Ende ist. Außerdem garantiert fgets(), dass die Zeichenkette mit '\0 ' beendet wird, deswegen wird auch ein Byte weniger eingelesen, als im Funktionsaufruf angegeben wird. Mit printf() kann dann der so gelesene Text ausgegeben werden. Nach dem vollständigen Auslesen muss die Datei mit fclose() geschlossen werden.

Neben fopen(), fclose(), feof() und fgets() gibt es noch eine Reihe sehr nützlicher Dateifunktionen, z.B. fprintf(), fgetc(), fputc(), fread(), fwrite(), etc.

- fprintf(*Datei*, *Formatstring*, *Werte*) gibt formatierte Werte wie fprintf() aus, aber nicht auf den Bildschirm sondern in eine *Datei*.
- fgetc(*Datei*) liest ein einzelnes Zeichen (0 bis 255) aus einer *Datei*.
- putc(*Zeichen*, *Datei*) schreibt ein Zeichen in eine *Datei*.
- fread(*Adresse*, *Anzahl*, *Größe*, *Datei*) schreibt *Anzahl*×*Größe* Bytes aus *Datei* und speichert sie ab *Adresse*.
- fwrite(*Adresse*, *Anzahl*, *Größe*, *Datei*) schreibt *Anzahl*×*Größe* Bytes ab *Adresse* in *Datei*.

Hilfe zu diesen Funktionen ist am einfachsten mit man zu erhalten.

Bei den obigen Funktionen zur Ein- und Ausgabe von Text kann statt einer Datei auch die Standardeingabe bzw. Standardausgabe als Quelle bzw. Ziel angegeben werden. Hierfür existieren standardmäßig der Eingabestrom stdin und der Ausgabestrom stdout vom Typ FILE* mit denen die Standardein- bzw. -ausgabe angesteuert werden können. Die bereits in Kapitel 1 vorgestellten Operationen zur Ein- und Ausgabe von Zeichen und Zeichenketten auf der Standardein- bzw. -ausgabe, sind also Spezialisierungen der oben vorgestellten Ein- und Ausgabefunktionen.

Ein weiterer Ausgabestrom ist `stderr`, auf dem in der Regel Fehlermeldungen des Programms ausgegeben werden. In diesem Zusammenhang ist auch die Funktion `exit(int)` zu erwähnen, mit der das Programm zu einem beliebigen Zeitpunkt beendet und an den Aufrufer ein enstprechender Fehlercode übergeben werden kann. Die typische Verwendung von `stderr` und `exit()` ist in folgendem Beispiel dargestellt.

```
#include <stdio.h>

int main(void)
{
   ...

   if (error)
   {
      /* Programm mit einer    *
       * Fehlermeldung beenden */
      fprintf{stderr,
        "Programmfehler!");
      exit(1);
   }

   ...

   /* regulaeres Programmende */
   return(0);
}
```

7.1.2 String-Operationen

Eigentlich kennt C keinen Datentyp für Strings, sondern höchstens Folgen von `char`s, die mit einem `'\0'`-Zeichen beendet werden. Für diese gibt es in `string.h` aber einige Funktionen, z.B. `strlen(`*Adresse*`)` zum Bestimmen der Länge der Zeichenfolge, die ab *Adresse* gespeichert ist. Andere String-Operationen in `string.h` sind `strcpy()` (zum Kopieren), `strcat()` (zum Zusammenfügen), `strcmp()` (zum Vergleichen) oder `strchr()` (zum Suchen nach einem Zeichen). Daneben gibt es auch in `stdlib.h` ein paar Funktionen für null-terminierte Zeichenfolgen, z.B. `atoi()` (zum Umwandeln eines Strings in ein `int`) oder `atof()` (zum Umwandeln eines Strings in ein `double`).

Als Beispiel hier ein Programm, das einen String mit durch Komma getrennten Ganzzahlen analysiert und die Zahlen ausgibt:

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

int main(void)
{
   char text[] = "42,007,137,0815";
   char *position;

   position = text;
   while (TRUE)
   {
      printf("%d\n",
         atoi(position));
      if (NULL ==
         strchr(position, ','))
      {
         break;
      }
      position =
         strchr(position, ',') + 1;
   }
   return(0);
}
```

Dabei wird in dem Zeiger `position` gespeichert, wo die nächste Zahl erwartet wird. Solange diese Position noch nicht über das Ende von `text` hinausweist, wird mit `atoi()` die Zeichenkette, auf die `position` zeigt, in eine Zahl umgewandelt und mit `printf()` ausgegeben. Dann wird das nächste Komma mit `strchr()` gesucht. Falls kein Komma mehr kommt, wird die Schleife beendet; sonst wird `position` auf das Zeichen *nach* dem nächsten Komma gesetzt.

7.1.3 Operationen für einzelne Zeichen

Neben Funktionen zur Manipulation von ganzen Zeichenketten stellt die Standardbibliothek auch Funktionen zur Untersuchung und Manipulation einzelner Zeichen zur Verfügung. Diese sind in der Header-Date `ctype.h` deklariert. Beispielsweise kann mit `isdigit(c)` überprüft werden, ob es sich bei einem Zeichen um eine Ziffer handelt, `isupper(c)` testet, ob das übergebene Zeichen ein Großbuchstabe ist. Mit `tolower(c)` und `toupper(c)` kann ein Zeichen in einen Klein- bzw einen Großbuchstaben gewandelt werden. `c` ist jeweils eine Variable vom typ int, die entweder eine ASCII-Zeichen repräsentiert oder den Wert `EOF` enthält. In Tabelle 7.2 sind einige Beispielfunktio-

Funktion	Erläuterung
`int isalnum(int c)`	alphanumerisches Zeichen?
`int isblank(int c)`	Leerzeichen?
`int isdigit(int c)`	Ziffer?
`int islower(int c)`	Kleinbuchstaben?
`int isupper(int c)`	Großbuchstaben?
`int tolower(int c)`	wandelt c in Kleinbuchstaben
`int toupper(int c)`	wandelt c in Großbuchstaben

Tabelle 7.2: Beispielfunktionen aus `ctype.h`

nen aus `ctype.h` aufgelistet.

7.1.4 Zeit- und Datumsfunktionen

Häufig kommt es auch vor, dass man die aktuelle Systemzeit oder das aktuelle Datum benötigt. Insbesondere bei zeitabhängigen Simulationen und zeitgesteuerten Animationen ist dies wichtig. Die Funktionen zur Abfrage der Uhrzeit bzw. des Datums sind weitgehend in der Datei `time.h` deklariert.

Die Funktion `time()` gibt die Anzahl der Sekunden zurück, die seit dem 1. Januar 1970 vergangen sind. Der Typ den `time()` zurückgibt, ist `time_t`, der eigentlich dem Typ `long` entspricht.

Da die wenigsten Menschen mit der Zahl der Sekunden seit dem 1.1.1970 umgehen können, gibt es hilfreiche Umrechnungsfunktionen. Mit der Funktion `localtime()` können Sie z.B. anhand einer Variablen vom Typ `time_t` eine Struktur namens `tm` füllen. Diese Struktur enthält Elemente für die Bestandteile des Datums und der Uhrzeit:

```
struct tm {
   /* Sekunden - [0,61] */
   int tm_sec;

   /* Minuten - [0,59] */
   int tm_min;

   /* Stunden - [0,23] */
   int tm_hour;

   /* Tag des Monats - [1,31] */
   int tm_mday;

   /* Monat im Jahr - [0,11] */
   int tm_mon;

   /* Jahr seit 1900 */
   int tm_year;

   /* Tage seit Sonntag - [0,6] */
   int tm_wday;

   /* Tage seit Neujahr - [0,365] */
   int tm_yday;

   /* Sommerzeit-Flag */
   int tm_isdst;
};
```

Möchte man eine genauere Zeitmessung durchführen, kann man die Funktiom `gettimeofday()` die im Header `sys/time.h` deklariert ist verwenden, die auch zusätzlich die vergangenen Mikrosekunden angibt. Sie hat folgende Syntax:

```
int gettimeofday(
  struct timeval *tv, void *tz);
```

Die Datenstruktur des ersten Parameters enthält die Sekunden und Mikrosekunden und ist folgendermaßen definiert:

```
struct timeval {
  /* Sekunden seit 1.1.1970 */
  time_t  tv_sec;

  /* Mikrosekunden */
  long    tv_usec;
};
```

Der zweite Parameter `tz` war ursprünglich für die Zeitzone vorgesehen. Allerdings hat sich die Umsetzung der verschiedenen Sommerzeitregeln als nicht so einfach erwiesen, sodass man als zweiten Parameter am besten 0 angibt.

7.1.5 Erzeugung von Zufallszahlen

Es gibt viele Algorithmen, die für ihre Durchführung zufällig erzeugte Zahlenwerte benötigen. Auch hierfür stellt die C-Standardbibliothek Funktionen zur Verfügung, die in `stdlib.h` deklariert sind. Mit `rand()` wird eine “Zufallszahl” zwischen 0 und `RAND_MAX` zurückgeliefert. Hierbei handelt es sich genauer gesagt nur um eine Pseudozufallszahl, d.h. bei wiederholtem Start eines Programms wird immer wieder die gleiche Folge von Zufallszahlen zurückgegeben.

Diesem Problem kann abgeholfen werden, indem man bei jedem neuen Start des Programms den Saatpunkt für die Berechnung der Pseudoufallszahlen neu setzt. Hierfür gibt es die Funktion `srand(unsigned int)`. Um immer wieder einen neuen Saatpunkt zu erhalten, wird häufig die Systemzeit verwendet. Ein möglicher Aufruf wäre z.B. `srand(time())`.

Möchte man Zufallszahlen erzeugen, die in einem anderen Bereich als von 0 bis `RAND_MAX` liegen, müssen die Rückgabewerte von `rand()` entsprechend skaliert werden. Die folgende Funktion gibt z.B. zufällige Gleitkommazahlen im Intervall $[0, 1]$ zurück:

```
float getProp()
{
  return((float)rand() / RAND_MAX);
}
```

7.1.6 Makro für die Fehlersuche

In der Header-Datei `assert.h` ist bei der Programmanalyse und Fehlersuche sehr hilfreiche Makro `assert()` definiert. Dies erhält als Argument einen logischen Ausdruck. Ist dieser nicht erfüllt, bricht die Programmausführung an dieser Stelle ab und es wird eine Fehlermeldung mit Angabe der Datei, der Funktion und der Zeilennumer, in der `assert` aufgerufen wurde, zurückgegeben.

Mit Hilfe vons `assert` ist es möglich Zusicherungen (englisch *assertions*) über den aktuellen Programmzustand zu machen. So kann z.B. zugesichert werden, dass der Parameter einer Funktion sich in einem bestimmten Wertebereich findet oder das Ergebnis eines Algorithmus gewissen Bedingungen entspricht. In der folgenden Beispielfunktion wird z.B. zugesichert dass der Übergabeparameter ungleich 0 ist, um somit eine Division durch 0 zu vermeiden.

```
float doSomeCalculations(float val)
{
  float result;
  assert(val != 0.0);
  ...

  result = 1.0 / val;
  ...
}
```

Wichtig ist, dass mit `assert` nur Programmierfehler abgefangen werden dürfen, d.h. Fehler, die z.B. aufgrund eines falsch implementierten Algorithmus entstehen. Fehler die jederzeit im laufenden Betrieb des Programms, z.B. durch falsche Benutzereingaben, verursacht werden können, dürfen nicht mit `assert` abgefangen werden, sondern müssen einer geeigneten Fehlerbehandlung unterzogen werden.

Damit durch die Verwendung des `assert`-Makros im späteren Betrieb keine Performanceeinbusen entstehen, kann dem Compiler das Argument `-DNDEBUG` mitgegeben werden. Dies führt dazu, dass durch das `assert`-Makro kein Code erzeugt wird, also auch der logische Ausdruck nie ausgewertet wird. Hierdurch wird nochmals klar, dass `assert` nur für das Aufdecken von Programmierfehlern verwendet werden darf. Außerdem darf der logische Ausdruck keinerlei Seiteneffekte auf den Programmzustand haben.

Kapitel 8

Einführung in die GLUT-Bibliothek

(Dieses Kapitel ist zum Teil eine Übersetzung von Mark Kilgard, *The OpenGL Utility Toolkit (GLUT) Programming Interface API*, `http://www.opengl.org/developers/documentation/glut/index.html`.)

Die GLUT-Bibliothek (OpenGL Utility Toolkit) wurde bereits in den vorangegangenen Beispielen und Aufgaben verwendet. Dabei wurde diese Hilfsbibliothek im wesentlichen nur dazu benutzt, ein Fenster zu öffnen, in welchem OpenGL-Zeichenoperationen durchgeführt werden konnten. Dementsprechend wurde bisher nur ein kleiner Teil der Funktionalität vorgestellt.

Tatsächlich erlaubt GLUT, OpenGL-Anwendungen mit mehreren Fenstern Plattform-unabhängig zu programmieren. Auch die Realisierung von Benutzerinteraktion über Maus, Tastatur und andere Eingabegeräte ist möglich. Es können Menüs definiert werden. Für einfache Graphikelemente wie Kugel, Würfel usw. stellt GLUT Hilfsfunktionen zur Verfügung.

Implementierungen der GLUT-Bibliothek existieren für X Windows (UNIX), MS Windows, OS/2. Obwohl für komplexere OpenGL-Applikationen in der Regel auf Plattform-abhängige aber mächtigere Schnittstellen zurückgegriffen wird, reicht die Funktionalität, die GLUT zur Verfügung stellt, für einfache Anwendungen voll aus.

Die Funktionen der GLUT lassen sich in unterschiedliche Gruppen einteilen, von denen die wichtigsten im Folgenden behandelt werden:

- Initialisierung
- Start der Eventverarbeitung
- Fenster-Management
- Menü-Management
- Callback Registrierung
- Zeichnen von Text
- Zeichnen geometrischer Formen
- State-Handling

8.1 Initialisierung

Bevor irgendeine GLUT-Funktion aufgerufen werden kann, muß zunächst die Bibliothek selbst initialisiert werden dies erfolgt durch Aufruf der Funktion:

```
void glutInit(int * argc,
              char **argv);
```

Er bewirkt, dass interne Datenstrukturen angelegt und eine Reihe von Zustandsvariablen auf sinnvolle Anfangswerte gesetzt werden. Außerdem wird überprüft, ob OpenGL auf der aktuellen Plattform unterstützt wird. Gegebenenfalls beendet der Aufruf das Programm mit einer Fehlermeldung.

Der Aufruf erwartet als Argument die Kommandozeile des Programms über die beiden Variablen `argc` und `argv`. Dadurch besteht die Möglichkeit beim Start eines GLUT-Programms Optionen für das Fenstersystem mit anzugegen z.B.:

`-geometry` **W**x**H**+**X**+**Y** Definiert die Standardgröße und -position des Anwendungsfensters.

`-iconic` Weist das Fenstersystem an, die Anwendung nach dem Start statt als Fenster als Icon darzustellen, welches sich erst nach dem Anklicken auf die Fenstergröße entfaltet.

`-gldebug` Dadurch wird die Anwendung angewiesen nach der Bearbeitung jedes Events den OpenGL-Fehlerstatus abzufragen und gegebenenfalls eine entsprechende Fehlermeldung auszugeben. Diese Funktion ist sehr nützlich um Programmierfehler zu finden.

`glutInit` untersucht, ob in der Kommandozeile die entsprechenden Optionen vorkommen. Dabei werden bekannte Optionen aus der Kommandozeile entfernt. Nach dem Aufruf enthält `argv` nur noch die Optionen, die `glutInit` nicht interpretieren konnte.

8.1.1 Anfangszustand

Eine Reihe von Zustandsvariablen kontrolliert den den Programmstart, die Anfangsgröße und -position des Anwendungsfensters sowie den Zeichenmodus. Tabelle 8.1 zeigt die Anfangswerte für diese Zustandsvariablen. Darüberhinaus sind die Namen der Funktionen enthalten, mit denen die entsprechenden Werte geändert werden können.

Variable	Anfangswert	Änderung mit
initWindowX	`-1`	`glutInitWindowPosition`
initWindowY	`-1`	`glutInitWindowPosition`
initWindowWidth	`300`	`glutWindowSize`
initWindowHeight	`300`	`glutWindowSize`
initDisplayMode	`GLUT_RGB` `GLUT_SINGLE` `GLUT_DEPTH`	`glutInitDisplayMode`

Tabelle 8.1: Die Anfangswerte bei Initialisierung der GLUT-Bibliothek

8.1.2 Graphikmodi

Durch den Graphikmodus wird angegeben, welche Eigenschaften der OpenGL-Kontext, der für das Zeichnen verwendet wird, besitzen soll. So lässt sich bestimmen, welche OpenGL-Buffer angelegt werden sollen. Der Graphikmodus wird mit der Funktion

```
void glutInitDisplayMode(
             unsigned int mode);
```

gesetzt. Einige der möglichen Parameter wurden bereits aufgeführt. Die Tabelle 8.2 gibt einen Überblick welche Werte `glutInitDisplayMode` als Argumente akzepiert. Die Parameter sind als Bitmaske implementiert. Das bedeutet, dass eine beliebige Kombination aus den Parametern durch logisches ODER erzeugt werden kann.

`GLUT_RGB` Bitmaske für den RGB-Modus. Der Framebuffer enthält je einen Wert für Rot, Grün und Blau.

`GLUT_RGBA|GLUT_ALPHA` Bitmaske für den RGBA-Modus. Siehe `GLUT_RGB`. Zusätzlich wird jedoch für jeden Pixel ein Transparentwert (Alpha) gespeichert. Die VerODERung mit dem zweiten Wert ist in diesem Fall unbedingt erforderlich, weil sonst kein Alpha-Kanal angelegt wird.

`GLUT_LUMINANCE` Bitmaske für den Luminanz-Modus. Hierbei gibt es nur einen Farbkanal je Pixel. Das bedeutet, dass die Darstellung mit unterschiedlichen Grauwerten erfolgt. Dabei wird für jedes Zeichenprimitiv nur der Rot-Kanal interpretiert und als Helligkeitswert gespeichert. Dieser Modus wird nicht von auf jeder Plattform unterstützt.

`GLUT_INDEX` Bitmaske für den Color-Index-Mode. Statt einer Farbe wird für jedes Graphik-Primitiv ein Index ($0 \ldots 255$) angegeben. Dieser wird für das Nachschlagen in einer Fabtabelle verwendet, in der die eigentlichen Farbwerte gespeichert sind. Auf diesen Modus wird hier nicht näher eingegangen.

`GLUT_SINGLE` Bitmaske für Single-Buffer.

`GLUT_DOUBLE` Bitmaske für Double-Buffer.

`GLUT_STEREO` Bitmaske für Stereo-Modus. Dabei werden abhängig von Single- oder Double-Buffer-Modus bis zu vier Versionen des Bildschirmspeichers gehalten. Dies erlaubt unterschiedliche Ansichten für das linke bzw. rechte Auge zu zeichnen, womit sich bei Verwendung einer Stereo-Hardware (z.B. Shutter-Glasses) eine räumliche Darstellung erzielen läßt.

`GLUT_ACCUM` Bitmaske für Accumulation-Buffer. Muss angegeben werden, wenn der Accumulation-Buffer verwendet werden soll.

`GLUT_DEPTH` Bitmaske für den Tiefenpuffer (Z-Buffer).

`GLUT_STENCIL` Bitmaske für den Stencil-Buffer.

Tabelle 8.2: Überblick über die Graphikmodi, die mit Hilfe von `glutInitDisplayMode` gesetzt werden können

8.2 Eventverarbeitung/Event-getriebene Programmierung

Um die Programmierung mit GLUT besser verstehen zu können, ist es wichtig zu erkennen, dass sich Programme mit graphische Benutzeroberfläche in ihrem Aufbau deutlich von sequentiellen Programmen unterscheiden.

8.2.1 Sequentielle Programmierung

Oftmals weisen C-Programme eine lineare Struktur auf. Das Programm erhält anfangs eine Menge von Eingabedaten, die es nach einem bestimmten Algorithmus verarbeitet. Am Ende steht die Ausgabe des berechneten Ergebnisses:

```
int main(int argc, char *argv[])
{
   /* Initialisierung */
   int min = atoi(argv[1]);
   int max = atoi(argv[2]);
   int i;

   double *ergebnis;
   ergebnis = (double *) malloc(
     (max-min+1)*sizeof(double));

   /* Berechnung */
   for (i = min; i < max; i++)
      ergebnis[i-min] = sqrt(i);

   /* Ausgabe */
   for (i = min; i < max; i++)
      printf("Wurzel %d = %lf\n",
             ergebnis[i-min]);

   /* Ende */
   return 0;
}
```

Hier erfolgt die Eingabe der Werte beispielsweise über Programmparameter. Eine andere Möglichkeit stellt das Einlesen der Daten aus einer Datei dar. Nach beendeter Berechung erfolgt die Ausgabe des Ergebnisses auf den Bildschirm. Diese Ausgabe kann natürlich auch innerhalb der Berechnung erfolgen.

8.2.2 Event-getriebene Programmierung

Im Gegensatz zu den sequentiellen Programmen sind Anwendungen mit graphischer Benutzeroberfläche Event-getrieben. Das bedeutet, dass diese Programme keinem vorgegebenen Programmablauf folgen sondern lediglich auf Ereignisse reagieren, die der Benutzer oder das Fenstersystem auslösen. Ereignisse, die der Anwender auslöst, sind beispielsweise ein Tastendruck oder das Klicken mit der Maus an eine Position innerhalb des Fensters. Das Fenstersystem löst unter anderem ein Ereignis aus, wenn das vorher verdeckte Anwendungsfenster wieder aufgedeckt wird. Das folgende Beispiel zeigt die typische Struktur eines Event-getriebenen Programms.

```
int main(int argc, char *argv[])
{
   /* Initialisierung */
   glutInit(&argc, argv);
   ...

   /* Event-Schleife */
   while(1)
   {
      ... /* warte auf Ereignis */
      ... /* bearbeite Ereignis */
   }

   /* Anweisungen hier werden
      niemals ausgefuehrt */
}
```

Auch das Event-getriebene Programm beginnt mit der Initialisierung seiner Daten, danach betritt es jedoch die sogenannte Event-Schleife (Eventloop), in der es auf Ereignisse wartet und in geeigneter Weise reagiert. Die Event-Schleife ist eine Endlosschleife, die das Programm niemals verlässt. Nachdem die Grundstruktur der Event-Schleife immer gleich ist, muß sie der Anwender in der Regel nicht selbst implementieren. Auch GLUT stellt eine fertig implementierte Schleife zur Verfügung.

8.2.3 glutMainLoop

... realisiert die Event-Schleife innerhalb der GLUT-Bibliothek. Ein GLUT-Programm ruft `glutMainLoop` auf, nachdem alle Initalisierungen wie das Anlegen der Fenster und der Menüs sowie Initialisierung selbst definierter Variablen erfolgt sind.

Der Aufruf von `glutMainLoop` sollte höchstens einmal in einem GLUT-Programm enthalten sein. Da das Programm die Event-Schleife nicht verlässt, kehrt der Aufruf von `glutMainLoop` niemals zurück. Anweisungen nach dieser Zeile werden nicht ausgeführt.

```
int main(int argc, char *argv[])
{
   /* Initialisierung */
   glutInit(&argc, argv);
   ...

   /* Event-Schleife */
   glutMainLoop();

   /* Anweisungen hier werden
      niemals ausgefuehrt */
}
```

8.3 Fensteroperationen

Um einen Bereich zu erhalten, in den gezeichnet werden kann, muss ein Fenster erzeugt werden. Dies erfolgt im Rahmen der Programminitialisierung vor dem Betreten der Event-Schleife. Die Funktion zum Erzeugen eines Fensters wurde bereits in den vorangegangenen Beispielen vorgestellt. Sie lautet

```
int glutCreateWindow(
        char * name);
```

Sie erzeugt ein Fenster mit der initialen Fenstergröße, die mit Hilfe von `glutInit` bzw. `glutInitWindowSize` gesetzt wurde, in der linken oberen Ecke des Bildschirms oder an der Position, die durch `glutInitWindowPos` definiert wurde. Dieses Fenster wird zum aktuellen Fenster. Als Rückgabewert liefert sie eine Fenster-ID, mit der das Fenster identifiziert werden kann. Die Fenster-ID des aktuellen Fensters kann mit

```
int glutGetWindow(void);
```

abgefragt werden. Mit

```
void glutSetWindow(int win);
```

kann mit Hilfe der Fenster-ID ein anderes Fenster zum aktuellen Fenster bestimmt werden.

Mit Hilfe von GLUT können verschiedene Modifikationen auf dem aktuellen Fenster durchgeführt werden. Diese sollen im Folgenden stichpunktartig aufgezählt werden.

`void glutPositionWindow(int x, int y);` ... ändert die Position des aktuellen Fensters. Die Werte haben nur Vorschlagscharakter, im Einzelfall kann das Fenstersystem Änderungen vornehmen, beispielsweise wenn die Position außerhalb des Bildschimbereichs liegt.

`void glutReshapeWindow(int width, int height);` ... ändert die Größe des aktuellen Fensters. Die Werte haben nur Vorschlagscharakter, im Einzelfall kann das Fenstersystem Änderungen vornehmen beispielsweise wenn die Größe den verfügbaren Bildschirmplatz überschreitet.

`void glutFullScreen(void);` ... ändert die Größe des Fensters so, dass es den gesamten verfügbaren Bildschirmplatz einnimmt.

Die Modifikationen wirken sich nicht sofort aus, sondern erst dann, wenn die Event-Schleife das nächste Mal durchlaufen wird. Dadurch können mehrere `glutPositionWindow` ... hintereinander ausgeführt werden. Ausschlaggebend ist der letzte der Aufrufe. So überschreiben auch `glutReshapeWindow`-Aufrufe nach `glutFullScreen` den Vollbildmodus wieder zugunsten der angeforderten Fenstergröße.

Auch die Umsetzung der folgenden Operationen ist abhängig vom Fenstersystem. Die Aufrufe wirken sich erst bei der nächsten Event-Schleife aus. Der jeweils letzte Aufruf ist ausschlaggebend.

`void glutPopWindow(void);` ... stellt das aktuelle Fenster in den Hintergrund.

`void glutPushWindow(void);` ... holt das aktuelle Fenster in den Vordergrund.

`void glutShowWindow(void);` ... zeigt das aktuelle Fenster wieder an, wenn es vorher verborgen war. (siehe `glutHideWindow`)

`void glutHideWindow(void);` ... versteckt das aktuelle Fenster.

`void glutIconifyWindow(void);` ... sorgt dafür, dass anstelle des eigentlichen Fensters nur ein kleines Icon erscheint. Beim Klicken auf dieses Icon erscheint dann wieder das Anwendungsfenster.

Neben diesen Funktionen, die sich auf Position, Größe und Erscheinungsbild auswirken, können noch zwei andere Parameter gesetzt werden.

`void glutSetWindowTitle(char * name);`
... definiert den Text, der in der Titelleiste des Fensters erscheint.

`void glutSetIconTitle(char * name);`
... definiert den Text, der unterhalb eines ikonifizierten Fensters dargestellt wird. Dieser Text kann sich von der eigentlichen Fenstertitelleiste unterscheiden.

`void glutSetCursor(int cursor);`
... ermöglicht es, die Gestalt des Maus-Cursors zu verändern. Die gewünschte Cursorform kann aus einer Reihe von Möglichkeiten ausgewählt werden. An dieser Stelle soll jedoch nicht näher auf die möglichen Parameter eingegangen werden. Zu diesem Zweck sei hier auf die GLUT-Dokumentation bzw. die GLUT-man-pages verwiesen.

Für das eigentliche Zeichnen mit OpenGL sind noch zwei weitere Funktionen wichtig.

`void glutPostRedisplay(void);` ... informiert die GLUT-Anwendung darüber, dass der Zeichenbereich neu gezeichnet werden muss. Dies kann beispielsweise dann erforderlich sein, wenn sich der Inhalt der Darstellung durch einen Animationsschritt geändert hat und neu gezeichnet werden muss. Das eigentliche Neuzeichnen wird dann an geeigneter Stelle von der GLUT-Bibliothek ausgelöst.

`void glutSwapBuffers(void);` Schaltet
im Double-Buffer-Modus zwischen den beiden Bildschirmspeichern um. Erst nach Aufruf dieser Funktion wird das gezeichnete Bild auch dargestellt. Er sollte deshalb immer am Ende der Zeichenfunktion stehen, wenn Double-Buffering verwendet wird.

8.4 Menüs

GLUT erlaubt die Definition von Menüs. Diese werden vom Fenstersystem in der Regel als sogenannte Popup-Menüs realisiert. Das bedeutet, dass das Menü normalerweise unsichtbar ist. Erst wenn der Benutzer mit der rechten Maustaste in das Anwendungsfenster klickt, klappt es auf. In manchen Umgebungen (z.B. unter MS Windows) kann das Menü auch in eine Menüzeile eingebunden sein.

Ähnlich wie bei den Fenstern kennt GLUT das Konzept des aktuellen Menüs. Alle Menüoperationen, die nicht explizit einen Identifikator für ein Menü erwarten, beziehen sich auf das aktuelle Menü. Ein Beispiel verdeutlicht die Programmierung eines Menüs:

Menüs werden mit

```
int glutCreateMenu(
      void (*func)(int value));
```

erzeugt. Das zuletzt erzeugte Menü ist immer das aktuelle. Es können beliebig viele Menüs erzeugt werden.

Nach der Erzeugung können dem aktuellen Menü beliebige Menüpunkte hinzugefügt werden. Dies erfolgt mittels

```
void glutAddMenuEntry(char *name,
                      int value);
```

Dieser Aufruf erhält zwei Argumente. Der erste ist ein String, der den Namen des Menüeintrags kennzeichnet. Der zweite ist ein beliebig gewählter Integer, der den Menüpunkt innerhalb des Menüs identifiziert. Menüpunkte werden sequentiell durchnumeriert. Der erste Eintrag eines Menüs trägt dabei die Nummer 1.

Das fertige Menü muss schließlich noch aktiviert werden. Dazu dient die Funktion

```
void glutAttachMenu(int button);
```

die als Argument denjenigen Maus-Button (`GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON` oder `GLUT_RIGHT_BUTTON`) erhält, bei dem das Menü erscheinen soll.

Die Reaktion auf einen ausgewälten Menüpunkt wird mit einer Callback-Funktion (ähnlich wie bei `glutDisplayFunc`) realisiert. Der Menü-Callback wird bei der Erzeugung des Menüs als Parameter übergeben. Jedesmal wenn der Benutzer einen Menüpunkt aktiviert, sorgt GLUT dafür, dass der entsprechende Menü-Callback aufgerufen wird. Dabei wird der Menüpunkt, der den Callback ausgelöst hat, als Parameter übergeben. In der Regel wird jedes Menü eine eigene Callback-Funktion besitzen. Es können jedoch auch mehrere Menüs dieselbe Funktion verwenden, wenn über die IDs der Menüpunkte eine eindeutige Identifikation gegeben ist.

8.4.1 Untermenüs

Untermenüs werden genauso erzeugt wie ein „Haupt"-Menü. Sie werden mit

```
void glutAddSubMenu(char *name,
                    int menu);
```

an das Ende des aktuellen Menüs angehängt. Auffällig ist, dass für diesen Menüpunkt kein Zweig in der Callback-Funktion vorgesehen ist. Dies ist nicht erforderlich, da GLUT die Funktionalität für das Aufklappen eines Untermenüs selbst implementiert.

```
int szene = 1;

void verarbeite(int value);
void zeigeSzene(int value);

void erzeugeMenue(void)
{
   /* Erzeuge Menue */
   int menuID = glutCreateMenu(
      verarbeite);
   glutAddMenuEntry("Neu", 7);
   glutAddMenuEntry("Laden", 2);
   glutAddMenuEntry("Speichern",
      5);

   /* Erzeuge ein Untermenue */
   int unterID = glutCreateMenu(
      zeigeSzene);
   glutAddMenuEntry("Szene A", 1);
   glutAddMenuEntry("Szene B", 2);
   glutAddMenuEntry("Szene C", 3);

   /* fuege Untermenue in
      Hauptmenue */
   glutSetMenu(menuID);
   glutAddSubMenu("Szene",
      unterID);

   /* lege Menue auf rechte
      Maustaste */
   glutAttachMenu(
      GLUT_RIGHT_BUTTON);
}

void verarbeite(int value)
{
   switch(value) {
      case 7:
         /* Anweisungen fuer
            "Neu" */
         break;
      case 2:
         /* Anweisungen fuer
            "Laden" */
         break;
      case 5:
         /* Anweisungen fuer
            "Speichern" */
         break;
      default:
         ...
   }
}

void zeigeSzene(int value)
{
   szene = value;
}
```

8.4.2 Sonstige Menü-Funktionen

Die übrigen GLUT-Menü-Funktionen sind selbsterklärend und werden hier nur der Vollständigkeit halber aufgezählt.

`void glutSetMenu(int menu);` ... setzt `menu` als aktuelles Menü.

`int glutGetMenu(void);` ... liefert die ID des aktuellen Menüs.

`void glutDestroyMenu(int menu);` ... löscht das Menü `menu`.

`void glutChangeToMenuEntry(int entry, char *name, int value);` ... ändert den Menüeintrag mit der Nummer `entry` auf einen neuen Wert.

`void glutChangeToSubMenu(int entry, char *name, int menu);` ... ändert den Menüeintrag mit der Nummer `entry` so, dass fortan der Menüpunkt das Untermenü `menu` aktiviert.

`void glutRemoveMenuItem(int entry);` ... löscht den Menüeintrag mit Index `entry`.

`void glutDetachMenu(int button);`
... entfernt das Menü, welches mit dem Maus-Button `button` verbunden ist.

8.5 Callbacks

Der eigentliche Programmablauf eines GLUT-Programms ist vor dem Programmierer verborgen. Er findet im Inneren der Event-Schleife statt. Wie aber kann der Programmierer auf Ereignisse des Benutzers oder des Fenstersystems reagieren? Ein in solchen Fällen oftmals verwendetes Konzept beruht auf dem Mechanismus der Callbacks, der auch von der GLUT-Bibliothek verwendet wird.

Für jedes potentielle Ereignis speichert GLUT einen Funktionszeiger. Diese Zeiger kann der Programmierer über spezielle GLUT-Funktionen setzen. Dieser Vorgang wird auch als Callback-Registrierung bezeichnet. Die Funktion, welche dabei registriert wird, heißt dementsprechend Callback. Die Callback-Funktionen können vom Programmierer frei gestaltet werden. Die einzige Randbedingung ist, dass die Signatur der Funktion dem entspricht, was GLUT an dieser Stelle erwartet.

Tritt ein Ereignis ein, so schaut GLUT nach, ob für dieses Ereignis ein Callback registiert wurde. Ist dies der Fall, wird die registrierte Funktion aufgerufen. Nach der Bearbeitung der Callback-Funktion setzt GLUT die Bearbeitung der Event-Schleife normal fort. Mit dem Rumpf der Callback-Funktion beschreibt der Programmierer somit die *Reaktion* auf bestimmte Ereignisse.

Ereignisse können global sein, wie das Ablaufen eines Timers, aber auch Kontext-gebunden, wie die Anforderung, dass ein bestimmtes Fenster neu gezeichnet werden soll. Auch die Größenänderung eines Fensters ist nur für dieses Fenster interessant. Daher speichert GLUT für Kontext-gebundene Ereignisse unterschiedliche Callbackfunktionen für jedes Fenster beziehungsweise Menü.

Die meisten Callbacks können auch wieder entfernt werden. Dazu wird anstelle des Funktionszeigers `NULL` übergeben. Dies funktioniert jedoch nicht bei der Zeichenfunktion `glutDisplayFunc`.

8.5.1 Übersicht über GLUT-Callbacks

Der wichtigste aller Callbacks wurde bereits vorgestellt. Es handelt sich um die Zeichenfunktion, die mit `glutDisplayFunc` registriert wird. Daneben gibt es noch eine ganze Reihe von anderen Callbacks.

`void glutDisplayFunc( void (*func)(void));` ... registriert eine Zeichenfunktion für das aktuelle Fenster. Der Aufruf `glutDisplayFunc` muß auf jeden Fall in einem GLUT-Programm enthalten sein, weil sonst überhaupt kein Fenster geöffnet wird.

Die Zeichenfunktion wird immer dann aufgerufen, wenn ein Neuzeichnen der Szene erforderlich ist. So beispielsweise, wenn das Fenster neu erzeugt wurde, wenn ein verdecktes Fenster aufgedeckt wird, wenn sich die Fenstergröße ändert oder wenn der Programmierer mittels `glutPostRedisplay` das Neuzeichnen verlangt.

`void glutReshapeFunc( void (*func)(int width, int height));` ... registriert eine Funktion, die aufgerufen wird, wenn sich die Größe eines Fensters ändert. So kann der Programmierer auf Änderungen reagieren.

Es existiert ein Standard-Callback für die Größenänderung. Dieser führt lediglich den OpenGL-Befehl `glViewport(0,0,width,height);` aus, was jedoch für ein einfaches Zeichenfenster vollkommen ausreicht.

`void glutKeyboardFunc( void (*func)(unsigned char key, int x, int y));` ... registriert eine Funktion, die aufgerufen wird, wenn der Anwender eine Taste betätigt. Neben dem ASCII-Code der Taste wird auch die Position des Mauszeigers (relativ zur linken oberen Ecke des Fensters) dem Callback als Parameter übergeben.

Sollen zusätzlich Modifier-Keys wie CTRL, ALT oder SHIFT interpretiert werden, so können die gedrückten Tasten mit Hilfe von `int glutGetModifiers(void)` ermittelt werden. Diese Funktion liefert eine logische VerODERung der Konstanten `GLUT_ACTIVE_SHIFT, GLUT_ACTIVE_CTRL` und `GLUT_ACTIVE_ALT`, je nachdem welche Tasten gerade gehalten werden.

Bei den Koordinaten `x` und `y` ist zu beachten, dass das X-Koordinatensystem eine andere Ausrichtung hat als das OpenGL-Koordinatensystem. X-Windows hat seinen Ursprung in der linken oberen Ecke. Die Werte für die x-Koordinate steigen nach rechts hin an, die Werte für die y-Koordinate steigen nach unten hin an (also genau umgekehrt zu OpenGL, wo die y-Werte nach oben hin ansteigen).

`void glutMouseFunc( void (*func)(int button, int state, int x, int y));` ... registriert eine Funktion, die aufgerufen wird, wenn der Benutzer die Maustaste im Inneren eines Fensters drückt oder losläßt. Somit werden beim Drücken und anschließendem Loslassen des Maus-Buttons zwei Ereignisse ausgelöst. Die gedrückte Taste wird über den Parameter `button` übergeben (`GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON`). Der Parameter `state` enthält die Information, ob der Mausknopf gedrückt (`GLUT_DOWN`) oder losgelassen (`GLUT_UP`) wurde.

Bei `GLUT_UP`-Ereignissen ist zu beachten, dass diese auch an Koordinaten auftreten können, die außerhalb des Anwendungsfensters liegen.

Gedrückte Modifier-Keys können wiederum mit `glutGetModifiers` abgefragt werden (siehe `glutKeyboardFunc`).

`void glutMotionFunc(void (*func)(int x, int y));` ... der hiermit registrierte Callback wird aufgerufen, wenn die Maus bei gedrückter Taste innerhalb des Fensters bewegt wird.

`void glutPassiveMotionFunc(void (*func)(int x, int y));` ... der hiermit registrierte Callback wird aufgerufen, wenn die Maus innerhalb des Fensters bewegt wird, ohne dass eine Taste gedrückt sein muss.

`void glutVisibilityFunc(void (*func)(int state));` ... der hiermit registrierte Callback wird aufgerufen, wenn sich die Sichtbarkeit eines Fensters ändert, beispielsweise wenn das Anwendungsfenster iconifiziert oder wieder aufgedeckt wird. Der Parameter `state` gib dabei an, ob das Fenster sichtbar geworden ist (`GLUT_VISIBLE`) oder unsichtbar (`GLUT_NOT_VISIBLE`).

`void glutEntryFunc(void (*func)(int state));` ... der hiermit registrierte Callback wird aufgerufen, wenn der Mauszeiger aus dem Fenster herausbewegt wird oder hineinbewegt wird. Dabei gibt die Variable `state` an, ob die Maus das Fenster gerade betreten (`GLUT_ENTERED`) oder verlassen (`GLUT_LEFT`) wurde.

`void glutSpecialFunc(void (*func)(int key, int x, int y));` ... der hiermit registrierte Callback funktioniert genauso wie der Callback für die Tastatur-Ereignisse mit der Ausnahme, dass dieser Callback ausgelöst wird, wenn eine Sondertaste auf der Tastatur betätigt wurde. Zu den Sondertasten zählen die F-Tasten `GLUT_KEY_F1 - GLUT_KEY_F12`, aber auch Cursortasten `GLUT_KEY_LEFT, GLUT_KEY_RIGHT, GLUT_KEY_UP, GLUT_KEY_DOWN, GLUT_KEY_PAGEUP, GLUT_KEY_PAGEDOWN, GLUT_KEY_HOME, GLUT_KEY_END, GLUT_KEY_INSERT`.

`void glutMenuStatusFunc(void (*func)(int status, int x, int y));` ... der hiermit registrierte Callback wird aufgerufen, wenn ein definiertes Menü aufpopt beziehungsweise wieder verschwindet. Die entsprechenden Werte für den Parameter `status` sind `GLUT_MENU_IN_USE` beziehungsweise `GLUT_MENU_NOT_IN_USE`. Mit Hilfe dieses Callbacks kann zum Beispiel eine Animation gestoppt werden, solange ein Menü angezeigt wird.

`void glutIdleFunc(void (*func)(void));` ... die hiermit registrierte Funktion wird aufgerufen, wenn das Programm sonst keine anderen Ereignisse bearbeiten muß. Dadurch eignet sich diese Funktion gut für Prozesse oder Bearbeitungsschritte, die im Hintergrund ablaufen sollen. Allerdings sollte der Rechenaufwand innerhalb der Idle-Funktion minimal gehalten werden, damit die Anwendung interaktiv bleibt.

`void glutTimerFunc(unsigned int msec, void (*func)(int value), int value);` ... hiermit kann ein Timer-Event generiert werden. Das Event wird nach mindestens `msec` Millisekunden ausgelöst, es kann aber auch abhängig von der Auslastung des Systems eine größere Zeitspanne verstrichen sein.

Timer-Events eignen sich besser als eine Idle-Funktion für die Animation einer Szene, weil auf diese Weise der Abstand der einzelnen Animationsschritte gleichmäßig ist.

Der Timer-Callback wird immer mit dem Wert als Argument aufgerufen, der bei der Registrierung als `value` übergeben wurde. Auf diese Weise kann zwischen mehreren Timer-Ereignissen unterschieden werden.

Der Timer-Event wird nur einmal ausgelöst. Wenn also eine kontinuierliche Folge von Timer-Events erzeugt werden soll, so muß dies durch kontinuierliches Aufsetzen eines neuen Timers erfolgen.

8.5.2 Animationen

Mit dem `IdleFunc`- bzw. `TimerFunc`-Callback lassen sich auf sehr einfache Weise Animationen erstellen. Diese sind ein wesentlicher Teil der Computergraphik, z.B. erlaubt erst die Drehung von dreidimensionalen Objekten ein wirkliches Verständnis der dreidimensionalen Struktur. Simulationen haben oft selbst einen Zeitparameter, so dass sie zur Darstellung eine Animation benötigen.

Ein Problem von Animationen ist die Zeit, die vom Löschen des Bildschirminhalts bis zum fertig aufgebauten Bild vergeht. Wenn der Bildschirminhalt in diesem Zeitraum dargestellt wird, sieht der Benutzer ein noch nicht fertig aufgebautes Bild, was sich dann meistens in einem hässlichen Flackern äußert. Deshalb kommt hier das bereits oben beschriebene Double-Buffering zum Einsatz, welches eigentlich recht einfach zu verstehen ist: Während ein Bild dargestellt wird, wird im Hintergrund (in einem zweiten Buffer) ein zweites Bild aufgebaut. Ist dieses zweite Bild fertig, wird dieser Buffer dargestellt, und der bisher dargestellte Buffer kann gelöscht und neu gefüllt werden, ohne dass der Benutzer eine Chance hat, vom Aufbau des Bildes etwas zu sehen, denn er sieht immer nur Buffer, in denen fertige Bilder stehen. Zum tauschen der Buffer muss am Ende des Zeichnens die GLUT-Funktion `glutSwapBuffers()` aufgerufen werden. Folgendes Beispiel zeigt, wie eine Animierte Grafik mit Hilfe des `IdleFunc`-Callbacks realisiert werden kann:

```
#include <stdio.h>
#include <math.h>
#include <GL/glut.h>

double winkel = 0.0;

void malen(void);
void idle(void);

int main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(
      GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize(400, 300);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Hallo!");
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 400.0, 0.0, 300.0,
      -1.0, 1.0);
   glutDisplayFunc(&malen);
   glutIdleFunc(&idle);
   glutMainLoop();
   return(0);
}

void malen(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_LINES);
      glColor3f(1.0, 1.0, 1.0);
      glVertex2i(200, 150);
      glVertex2f(200.0 + 150.0 *
         cos(winkel), 150.0 -
         150.0 * sin(winkel));
   glEnd();
   glFlush();
   glutSwapBuffers();
}

void idle(void)
{
   winkel = winkel + 0.01;
   glutPostRedisplay();
}
```

Wichtig ist, dass die Funktion `idle()` nur den Winkel, um den die gezeichnete Linie gedreht werden soll ändert. Dieser wird in einer externen Variablen gespeichert und ist so in allen Funktionen sichtbar. Das eigentliche Zeichnen wird dann über die Funktion `glutPostRedisplay` angestoßen.

Der `IdleFunc`-Callback ist nur eingeschränkt für Animationen geeignet, da der Abstand zwischen zwei Animationsschritten nicht fest ist, sondern immer davon abhängt, welche anderen Ereignisse noch bearbeit werden müssen. Deshalb eignet sich hier der `TimerFunc`-Callback besser, wobei auch hier die Zeitspanne abhängig von der Systemauslastung nicht unbedingt fest ist. Bei Verwendung des `TimerFunc`-Callbacks ergeben sich folgende Änderungen in obigem Progamm:

```
int main(int argc, char **argv)
{
   ...
   /* Starte Timer erstmals */
   glutTimerFunc(20, &tick, 0);
```

```
}

void tick(int value)
{
   /* Starte Timer erneut */
   glutTimerFunc(20, &tick, 0);

   winkel = winkel + 0.01;
   glutPostRedisplay();
}
```

8.6 Zeichnen von Text

OpenGL stellt keine direkte Funktionalität zur Verfügung, um einen Text in ein OpenGL-Fenster zu zeichnen. Vielmehr muss der Programmierer für jedes Zeichen des Zeichensatzes ein eigenes Bitmap erzeugen. Mit Hilfe dieser Bitmaps kann dann Text in einem OpenGL-Fenster ausgegeben werden, indem die Bitmaps der einzelnen Zeichen mit dem korrekten Abstand hintereinander gesetzt werden.

Die GLUT-Bibliothek erleichtert diesen Vorgang, indem sie bereits eine Reihe solcher Bitmap-Fonts zur Verfügung stellt und auch das Zeichnen der Buchstaben mit entsprechenden Funktionen unterstützt.

Darüberhinaus stellt GLUT auch sogenannte Stroke-Fonts zur Verfügung. Hierbei wird anstelle eines Bitmaps der Umriß jedes Buchstabens in Form eines Linienzuges gezeichnet. Die folgende Funktion demonstriert die Textausgabe in ein OpenGL-Fenster:

```
void text(void *font, int x, int y,
          char *string)
{
   int len,i;

   len = (int)strlen(string);

   /* Positioniere Zeichenmarke */
   glRasterPos2i(x, y);

   for(i = 0; i < len; i++)
   {
      glutBitmapCharacter(
          font, string[i]);
   }
}
```

Zunächst muss die gewünschte Position `(x, y)`, an der der Text erscheinen soll, als aktuelle Zeichenposition gesetzt werden. Dies erfolgt mit dem OpenGL-Kommando `glRasterPos2i`. Danach wird der auszugebende Text zeichenweise bearbeitet und als Bitmap an die jeweils aktuelle Stelle gezeichnet. Dabei verändert `glutBitmapCharacter` bei jedem Aufruf entsprechend der Breite des aktuellen Zeichens die Rasterposition, so dass keine Neupositionierung erforderlich ist.

Dennoch stellt GLUT Funktionen zur Verfügung, mit denen die Breite eines einzelnen Zeichens ermittelt werden kann:

```
int glutBitmapWidth(
       GLUTbitmapFont font,
       int character);
```

Analog existieren für das Zeichnen im Stroke-Modus innerhalb der GLUT die Funktionen:

```
void glutStrokeCharacter(
       void * font,
       int character);
int glutStrokeWidth(
       GLUTbitmapFont font,
       int character);
```

Für das Zeichnen von Text stehen unterschiedliche Zeichensätze zur Verfügung, die mit entsprechenden Konstanten beim Aufruf der Zeichenfunktionen übergeben werden. Tabelle 8.3 gibt einen Überblick über die möglichen Fonts.

Konstante	Breite × Höhe	Schriftfamilie
GLUT_BITMAP_8_BY_13	8×13	Standard Fixed
GLUT_BITMAP_9_BY_15	9×15	Standard Fixed
GLUT_BITMAP_TIMES_ROMAN_10	prop. 10pt	TimesRoman
GLUT_BITMAP_TIMES_ROMAN_24	prop. 24pt	TimesRoman
GLUT_BITMAP_HELVETICA_10	prop. 10pt	Helvetica
GLUT_BITMAP_HELVETICA_12	prop. 12pt	Helvetica
GLUT_BITMAP_HELVETICA_18	prop. 18pt	Helvetica

Tabelle 8.3: Die von der GLUT-Bibliothek unterstützten Zeichensätze.

8.7 Zeichnen geometrischer Objekte

OpenGL unterstützt nur einfache Basisprimitive wie Punkt, Linie und Polygon, zum Zeichnen geometrischer Objekte. Sollen andere Objekte wie ein Würfel oder eine Kugel gezeichnet werden, so muss dieses Objekt aus lauter Linien oder Polygonen aufgebaut werden. Dies

mag für einen Würfel noch einfach sein. Schon die Polygonalisierung einer Kugel erfordert jedoch einen gewissen Aufwand.

Die GLUT-Bibliothek unterstützt den Programmierer bei der Modellierung einfacher dreidimensionaler Objekte, indem sie eine Reihe von Hilfsfunktionen zur Verfügung stellt, mit der solche Objekte gezeichnet werden können. Im Folgenden sind alle unterstützten Objekte aufgezählt. Die Funktionen sind im wesentlichen selbsterklärend. Die geometrischen Primitive werden mittig um den Nullpunkt gezeichnet.

```
glutSolidSphere(GLdouble radius,
                GLint slices,
                GLint stacks);
glutWireSphere(GLdouble radius,
               GLint slices,
               GLint stacks);

glutSolidCube(GLdouble size);
glutWireCube(GLdouble size);

glutSolidCone(GLdouble base,
              GLdouble height,
              GLint slices,
              GLint stacks);
glutWireCone(GLdouble base,
             GLdouble height,
             GLint slices,
             GLint stacks);

glutSolidTorus(GLdouble inRadius,
               GLdouble outRadius,
               Glint nsides,
               GLint rings);
glutWireTorus(GLdouble inRadius,
              GLdouble outRadius,
              Glint nsides,
              GLint rings);

glutSolidDodecahedron(void);
glutWireDodecahedron(void);

glutSolidOctahedron(void);
glutWireOctahedron(void);

glutSolidTetrahedron(void);
glutWireTetrahedron(void);

glutSolidIcosahedron(void);
glutWireIcosahedron(void);

glutSolidTeapot(GLdouble size);
glutWireTeapot(GLdouble size);
```

8.8 State-Handling

Ebenso wie OpenGL speichert die GLUT-Bibliothek eine ganze Reihe von internen Zuständen, um die Anzahl der Parameter für die Funktionen möglichst gering zu halten. Der Wert der einzelnen Zustandsvariablen kann mit der Funktion:

```
int glutGet(GLenum state);
```

ermittelt werden. Eine Aufzählung aller Zustandsvariablen würde hier zu weit führen. Der geneigte Leser wird deshalb auf die GLUT-Dokumentation beziehungsweise die GLUT-man-pages verwiesen. Ebenso sollen die beiden Funktionen

```
int glutDeviceGet(GLenum info);
int glutExtensionSupported(
              char *extension);
```

mit denen die angeschlossenen Eingabegeräte beziehungsweise die von der Graphikhardware unterstützten OpenGL-Erweiterungen erfragt werden können, nur dem Namen nach erwähnt werden.

Übungen zum vierten Tag

Aufgabe IV.1

Das Logo der Universität Stuttgart besteht aus 133 auf der Spitze stehenden Quadraten. Die Mittelpunkte dieser Quadrate sind als Punktpaare in der Textdatei `unilogo.dat` gespeichert die Sie unter http://www.vis.uni-stuttgart.de/ger/teaching/lecture/CKompaktkurs/unilogo.dat herunterladen können. Schreiben Sie ein Programm das diese Textdatei einliest und das Logo graphisch darstellt. Zeichnen Sie dazu an jedem dieser Punkte ein auf der Spitze stehendes Quadrat mit der Kantenlänge 0.9259259.

Programmieren Sie eine animierte Darstellung des Logos, bei der sich die Größe der Quadrate zyklisch ändert, so dass ein „pumpender“ Effekt entsteht. Versuchen Sie insbesondere durch die Verwendung von Funktionen die Größe Ihres Programms möglichst klein zu halten.

Aufgabe IV.2

Es kommt häufig vor, dass man alten, längst vergessenen Code, wiederverwenden will. In dieser Übung soll der Code zur Berechnung der Mandelbrot-Menge aus Aufgabe II.3 reaktiviert werden.

Öffnen Sie zwei Fenster und stellen Sie in einem die Mandelbrot-Menge dar. Im zweiten soll eine sogenannte Julia-Menge dargestellt werden, die nach folgender Vorschrift berechnet wird:

Julia-Mengen (nach ihrem Entdecker, Gaston Julia, benannt) werden ganz ähnlich wie Mandelbrotmengen berechnet, mit dem Unterschied, dass c_x und c_y für das ganze Bild konstant sind (und frei vorgegeben werden können) und sich der Startwert $\begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$ aus den skalierten und verschobenen Koordinaten eines Bildpunktes ergibt. Es soll etwa gelten: $-2 < x_0 < 2$ und $-2 < y_0 < 2$.

Mit Hilfe einer Callback-Funktion soll nun die Julia-Menge neu berechnet werden, sobald der Benutzer auf einen Punkt im Fenster der Mandelbrot-Menge klickt. Die Koordinaten des angeklickten Punkts sollen dabei die Werte (c_x, c_y) für die Berechnung der Julia-Menge angeben.

Aufgabe IV.4

Eine Beispiel für Iterationen in zwei Dimensionen ist das sogenannte *Chaos Game* von Michael F. Barnsley, bei dem zufällig eine von mehreren (linearen) Iterationsvorschriften angewendet wird. Und zwar mit Wahrscheinlichkeit 0.02 die Vorschrift

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.27 y_n \end{pmatrix},$$

mit Wahrscheinlichkeit 0.15 die Vorschrift

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} -0.139 x_n + 0.263 y_n + 0.57 \\ 0.246 x_n + 0.224 y_n - 0.036 \end{pmatrix},$$

mit Wahrscheinlichkeit 0.13 die Vorschrift

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 0.17 x_n - 0.215 y_n + 0.408 \\ 0.222 x_n + 0.176 y_n + 0.0893 \end{pmatrix},$$

und mit Wahrscheinlichkeit 0.7 die Vorschrift

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 0.76\cos(\phi) x_n + 0.76\sin(\phi) y_n + 0.1075 \\ -0.76\sin(\phi) x_n + 0.76\cos(\phi) y_n + 0.27 \end{pmatrix},$$

Der Startpunkt $\begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$ soll dabei $\begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$ sein. ϕ ist ein frei wählbarer kleiner Winkel.

Schreiben Sie ein Programm, das die Iteration 10000-mal durchführt und nach jedem Iterationschritt einen Punkt an der Position (x_n, y_n) zeichnet. Skalieren und verschieben Sie die Fensterkoordinaten geeignet, so dass alle Punkte sichtbar sind.

Animieren Sie die Ausgabe ihres Programms, indem Sie den Wert des Winkels ϕ mit Hilfe der Maus einstellbar machen.

Aufgabe IV.4

Animieren Sie 100 Punkte in einem Fenster. Speichern Sie dazu zu jedem Punkt sowohl seine zweidimensionalen Koordinaten $\vec{r}_i$ mit $0 \leq i < 100$, als auch seine Geschwindigkeit $\vec{v}_i$. Starten Sie mit ruhenden Punkten an unterschiedlichen Positionen in der Nähe des Ursprungs, der sich in der Mitte des Fensters befinden sollte. Das Koordinatensystem sollte außerdem so gewählt werden, dass die Seitenlänge des Fensters in diesen Koordinaten etwa 200 ist.

Berechnen Sie in in jedem Zeitschritt der Länge d, die neuen Positionen aller Punkte nach den Formeln

$$\begin{aligned} \vec{r}_i &\leftarrow \vec{r}_i + \vec{v}_i d + \frac{1}{2}\vec{a}_i d^2 \\ \vec{v}_i &\leftarrow 0.999 \cdot (\vec{v}_i + \vec{a}_i d) \end{aligned}$$

mit

$$\begin{aligned}\vec{a}_i &= \vec{r}_i \left(\frac{1}{2500} \sin(\vec{r}_i^{\,2}/1000) - \frac{1}{10000} \exp(\vec{r}_i^{\,2}/1000) \right) \\ &\quad \pm 100 \cdot (\vec{m} - \vec{r}_i) \exp\left(-(\vec{m} - \vec{r}_i)^2/100\right)\end{aligned}$$

und den Mauskoordinaten $\vec{m}$ (im OpenGL-Koordinatensystem!). Wählen Sie d geeignet. Für $\pm$ kann $+$ oder $-$ eingesetzt werden.

Teil V

Fünfter Tag

Kapitel 9

Entwicklungswerkzeuge

In Kapitel 4.3.2 haben wir bereits `make` als ein sehr hilfreiches Tool für die Übersetzung von C-Programmen kennengelernt. In diesem Kapitel sollen nun weitere Entwicklungswerkzeuge vorgestellt werden, die das Leben des Programmierers erheblich vereinfachen können.

9.1 Debugger

Die meisten Entwicklungsumgebungen enthalten ein spezielles Programm namens Debugger zum Aufspüren von Fehlern („bugs“; der Ausdruck kommt angeblich aus der Frühzeit des Informationszeitalters als noch allerhand Kleingetier für Fehler in Computern sorgte und mühsam von Hand entfernt werden musste). Ein Debugger dient vor allem dazu, ein Programm kontrolliert (schrittweise) auszuführen, so dass fehlerhafter Code schneller lokalisiert werden kann.

Der Standard-Debugger der Gnu-Entwicklungsumgebung heißt `gdb` (Gnu-DeBugger) und wird mit `gdb` *Programmdatei* [*core-Datei*] aufgerufen. Die (optionale) core-Datei ist eine Art Bericht des Betriebssystems nach einem Programmabsturz, der meistens in einer Datei namens `core` abgespeichert wird und dem Programmierer dazu dienen soll, die näheren Umstände des Absturzes zu erforschen. (Das Schreiben dieses Berichts, der mehr oder weniger den gesamten Systemzustand zum Zeitpunkts des Absturzes enthält, wird auch als core dump bezeichnet.)

Das erste Problem beim Debuggen eines Programms ist, dass die ausführbare Programmdatei normalerweise nur noch Informationen enthält, die für den Computer zur Ausführung des Programms relevant sind. Damit sind die meisten für menschliche Programmierer interessanten Informationen wie Symbolnamen oder die ursprünglichen C-Befehle ausgeschlossen. Damit der Compiler auch diese Informationen in die ausführbare Programmdatei schreibt, muss in der Regel ein entsprechendes Compiler-Flag gesetzt werden; beim `gcc` ist dies `-ggdb`. (`-g` alleine schreibt (meistens weniger) Debugging-Information im „nativen“ Format der jeweiligen Plattform in die Programmdatei, so dass sie auch mit dem „ nativen“ Debugger ausgewertet werden kann.)

Wenn der `gdb` mit einer Programmdatei aufgerufen wird, die Debugging-Information enthält, passiert erstmal gar nichts, weil der `gdb` auf Kommandos wartet. Hier sind ein paar der nützlichsten:

- `quit` beendet den `gdb`.
- `help` [*gdb-Kommando oder -Thema*] gibt eine Beschreibung von gdb-Kommandos bzw. -Themen aus.
- `break` [*Dateiname*:]*Funktionsname* oder `break` [*Dateiname*:]*Zeilennummer* setzt einen Haltepunkt (breakpoint) an der angegebenen Zeilennummer bzw. beim Einsprung in die angegebene Funktion.
- `delete` löscht alle Haltepunkte.
- `run` startet ein Programm zum Debuggen. (Im Allgemeinen sollte vorher mindestens ein Haltepunkt gesetzt worden sein, z.B. bei der Funktion `main`.)
- `continue` lässt ein Programm nach Erreichen eines Haltepunkts weiterlaufen.
- `step` führt eine C-Anweisung aus und springt gegebenenfalls in Funktionen hinein.
- `next` führt eine C-Anweisung aus, aber springt nicht in Funktionen.

- `finish` führt ein Programm bis zum Rücksprung aus der aktuellen Funktion aus.
- `print` *Ausdruck* gibt einen C-Ausdruck aus, der z.B. auch Variablen enthalten darf.
- `backtrace` gibt eine Beschreibung des Callstacks (also aller derzeit aktiven Unterprogrammaufrufe) aus.

Natürlich gibt es noch viel mehr gdb-Kommandos, aber fürs erste Ausprobieren sollte das reichen.

Wem das spartanische Kommandozeilen-Interface des `gdb` nicht zusagt, kann meistens auch auf Debugger mit graphischer Benutzungsoberfläche wie `ddd` zurückgreifen. Deren Bedienung ist meistens mehr oder weniger selbsterklärend. (Jedenfalls dann, wenn man schon mal mit einem anderen Debugger gearbeitet hat.)

9.2 Analysetools

Eine Programmiersprache wie C enthält viele Fallstricke, die zwar zu gültigem Code führen, aber nicht den vom Programmierer beabsichtigten Effekt haben. Warnungen des C-Compilers können auf viele derartige Probleme hinweisen, aber es gibt Programme, die den Sourcecode noch wesentlich genauer nach möglichen Fehlern durchsuchen. Ein Beispiel ist das Tool `lint`, das z.B. mit `lint` *.c-Datei* aufgerufen wird

Aber nicht nur der Sourcecode kann genauer analysiert werden: Sogenannte Profiler, z.B. `gprof` analysieren den tatsächlichen Programmablauf, insbesondere um festzustellen, wo der Computer die meiste Zeit „verbrät“. Damit kann bei zeitkritischen Programmen festgestellt werden, welche Programmteile noch optimiert werden sollten, bzw. welche Optimierungen überflüssig sind.

9.3 Sourcecode-Management

Sobald mehrere Entwickler an einem Projekt arbeiten wollen, bzw. ein einzelner (zerstreuter) Entwickler immer wieder Zugriff auf alte Programmversionen braucht, sollte ein Sourcecode-Management-System wie RCS oder (das mächtigere) CVS benutzt werden. Ein neueres System ist Subversion, dass immer häufiger bei Open-Source-Projekten CVS als Standardversionsverwaltung ablöst.

9.4 Dokumentationssysteme

Mit der Größe eines Projekts wird auch die Dokumentation des Sourcecodes immer wichtiger. Dabei können Tools zur „automatischen“ Dokumentationserstellung (z.B. doxygen) gute Dienste leisten. Die meisten dieser Tools erwarten, dass Programmierer speziell formatierte Kommentare in den Sourcecode einfügen, die dann zusammen mit der automatischen analysierten Programmstruktur in nett formatierte Dokumentationstexte (typischerweise in HTML, LaTeX, PostScript, etc.) zusammengefasst werden. Sie biueten zudem den Vorteil, dass sie automatisch einen bestimmten Stil für die Kommentare vorgeben und so auch der Quellcode besser lesbar wird.

9.5 Integrierter Entwicklungsumgebungen

Wie man sieht, gibt es eine Vielzahl von unabhängigen Werkzeugen, die den Programmierer bei der Softwareentwicklung unterstützen. Hier kann man jedoch sehr leicht den Überblick verlieren und der Aufwand erscheint oft höher als der Nutzen. Diesen Misstand versuchen die sogenannten integrierten Entwicklungsumgebugnen, kurz IDEs (*Integratet Development Environments*), zu beheben, indem sie möglichst viele dieser Tools in einer einzelnen Umgebung zusammenfassen. Ausserdem bieten Sie meist Hilfsmittel wie Syntaxhighlighting und automatische Codevervollständigung an.

Typische Vertreter dieser Programme sind z.B. KDevelop für Linux und Dev-C++ in Kombination mit dem MinGW-Compiler für Windows. Diese unterstützen zwar schwerpunktmäßig die objektorientierte Programmentwicklung mit C++, sind aber prinzipiell auch für die C-Programmierung geeignet. Auch die klassischen Editoren wie Emacs und vi werden durch entsprechende Einstellungen und Erweiterungen zu sehr mächtigen Entwicklungswerkzeugen.

Kapitel 10

Stilvolles C

(Dieses Kapitel ist zum Teil ein übersetzter Auszug aus David Straker, *C-Style: Standards and Guidelines*, Prentice Hall, 1992.)

10.1 Was sind Kodierung-Standards?

Die Programmiersprache C hat eine definierte Syntax, die definiert, was ein gültiges C-Programm ist, und was nicht. Trotz dieser Regeln bleibt vieles offen: Wie sollen Variablen, Funktionen, Dateien etc. genannt werden, was und wie kommentiert wird, wie Text eingerückt wird, wo Leerzeichen stehen, etc. Kodierung-Standards enthalten Regeln, die einige dieser offenen Fragen entscheiden. In der Praxis werden diese Regeln aber oft nicht erzwungen; entsprechend sind es eigentlich Richtlinien *(Guidelines)*.

10.2 Wofür sind C-Kodierung-Standards gut?

Es scheint schwierig genug zu sein, korrekten und effizienten C-Code zu schreiben. Der C-Compiler erscheint dabei manchmal mehr als fieser Gegner, der die Programmiererin mit unverständlichen Fehlermeldungen bombadiert, als das unglaublich mächtige Tool, das er eigentlich ist. Wenn sich in diesem Kampf Programmiererin gegen Compiler der Compiler nicht um Kommentare, Leerzeichen und Tabulatoren, Namen von Variablen, etc. kümmert, warum sollte es die Programmiererin tun? Wofür sollten Kodierung-Standards, die nur noch mehr Regeln zur Kodierung (zusätzlich zur C-Syntax) enthalten, gut sein?

David Straker gibt einige Ziele an, die mit Hilfe von Kodierung-Standards besser erreichbar sind:

- **Produktivität:** Dazu sollte Code
  - geeignet für Entwicklungstools,
  - portabel und
  - wiederverwendbar sein.
- **Qualität:** Code sollte Spezifikationen entsprechen. Deshalb sollte Code
  - überprüfbar und
  - verlässlich sein.
- **Wartbarkeit:** Dazu sollte Code
  - erweiterbar und
  - selbst-diagnostizierend sein.
- **Verständlichkeit:** Dazu sollte Code
  - konsistent und
  - kommunizierbar sein.

10.3 Allgemeine Grundsätze

- **Denke an den Leser!**
- **„Keep it Simple, stupid! ;-)“** (KISS)
- **Sei explizit!**
- **Sei konsistent!**
- **Nimm den kleinstmöglichen Bereich!** (Sichtbarkeits- und Gültigkeitsbereiche von Variablen, Funktionen, etc.)

- Die letzten vier Punkte lassen sich zu **SECS** zusammenfassen: **Simple, Explicit, Consistent, minimal Scope**
- Dokumentiere den eingesetzten Standard!
- Benutze Standard-Bibliotheken!
- Benutze Entwicklungstools: Editoren mit Syntax-Highlighting, Compiler mit vielen Warnungen (`gcc -Wall -ansi -pedantic ...`), Suchprogramme, language checkers (vor allem `lint`), complexity checkers, pretty-printers, ...

10.4 Kommentare

Allgemein sollten Kommentare

- klar und vollständig sein;
- kurz sein, um zu sagen *was* passiert, und länger, um zu sagen *warum*;
- korrekt zein;
- vom Code unterscheidbar sein und
- in Größe und Ausführlichkeit der Wichtigkeit und Komplexität des beschriebenen Codes entsprechen.

Kopfkommentare *(heading comments)*:

- ... sollten am Anfang jeder Datei, jeder Funktionsdefinition und vor größeren Datendefinitionen stehen.
- ... sind strukturiert und bestehen aus mehreren Zeilen.
- ... können auch mit manchen Entwicklungstools bearbeitet werden.
- ... sollten auch den Kontext (auch historisch) einer Datei oder Funktion beschreiben.
- ... sollten möglichst viel Hilfe, Warnungen, Tips, Annahmen und Grenzen des Codes enthalten.

Blockkommentare

- ... stehen nicht mit Code in einer Zeile.
- ... beschreiben den vorhergehenden und/oder nachfolgenden Code.
- ... sollten mit Hilfe von Leerzeilen Codezeilen zugeordnet werden.
- Alle Zeilen von mehrzeiligen Blockkommentaren sollten mit `/*` oder `*` beginnen.

Zeilenkommentare

- ... stehen mit Code in einer Zeile.
- ... sollten vor allem in besonderen Situationen verwendet werden.

Datenkommentare:

- ... sollten bei alle Datendeklarationen stehen.
- ... sollten sehr genau sein und den Zweck von Daten beschreiben.
- ... sollten auch Beispiele für die Verwendung und mögliche Werte geben.

10.5 Namen

- ... sollten auch international lesbar sein (also: englisch).
- Namen sollten nicht zu lang sein.
- Um Namen zu kürzen sollten Abkürzungen verwendet werden.
- Abkürzungen sollten konsistent sein. Eine Liste mit Standardabkürzungen ist nützlich.
- Kurze Namen sollten nur in kleinen Bereichen verwendet werden, und auch dann nur, wenn ihre Bedeutung offensichtlich ist.
- Wörter in Namen sollten konsistent getrennt sein (`einName`, `ein_name`, `ein_Name`).
- Namen sollten der korrekten Rechtschreibung entsprechen.
- Namen sollten eindeutig sein.
- Unterstriche sollten nicht in Paaren und nur innerhalb von Namen auftreten.
- Vorsicht bei Ziffern, die wie Buchstaben aussehen (`l` und `1`).
- Prozedurnamen sollten Verb-Nomen-Kombinationen sein.

- Gruppen von Funktionen, Konstanten, etc. sollten ein gemeinsames Prefix haben.
- Variablennamen sollten aus Nomen bestehen.
- Boolsche Variablennamen sollten mit `Is` oder `is` (ist ...?) beginnen.
- Mit `#define` vereinbarte Symbole sollten in Großbuchstaben geschrieben sein.

10.6 Layout

Leerzeichen:

- Kein Leerzeichen bei den binären Operator mit höchster Priorität: `a(b)`, `a[b]`, `a.b`, `a->b`.
- Kein Leerzeichen nach unären Operatoren (und vor Postfix-De/Inkrementoperator).
- Leerzeichen vor und nach allen anderen binären und ternären Operatoren.
- Kein Leerzeichen vor `,` und `;`, aber danach ein Leerzeichen.
- Leerzeichen vor und nach Keywords.
- Wenn (ausnahmsweise) nach `(` ein Leerzeichen, dann auch vor dem entsprechenden `)`.

Zeilenumbruch:

- In jeder Zeile sollte nur eine Anweisung stehen.
- Bei mehrzeiligen Anweisungen sollte die zweite und nachfolgende Zeilen eingerückt werden.
- Ausdrücke sollten nach Regeln umgebrochen werden (z.B. nach `,` und `;` umbrechen wenn möglich).

Klammern und Blöcke:

- Einrückungen von Blöcken sollten konsistent sein.
- Nach `if`, `else`, `while`, `for`, `switch`, `do` immer einen Block, auch wenn nur eine Anweisung.
- Zu tiefe (mehr als 3) Verschachtelungen sollten vermieden werden.

10.7 Datei-Layout

Allgemein:

- Dateien sollten strukturiert sein. Dazu ist ein Template nützlich.
- Reihenfolge von Einträgen: extern vor intern, öffentlich vor privat, alphabetisch.
- Kopfkommentar mit Dateinamen, Autor(in), Kurzbeschreibung, etc.
- Länge sollte auf 1000 Zeilen (20 Seiten), Breite auf 80 Zeichen beschränkt sein.

Header-Dateien (`.h`-Dateien):

- Schutz vor mehrfachen `#includes` mit `#if !defined ..._H`.
- Keine Speicherplatzreservierungen in Headerdateien. (Keine Definitionen, nur Deklarationen.)
- Möglichst pro `.c`-Datei eine `.h`-Datei.
- Funktionsprototypen von allen öffentlichen Funktionen.
- Deklarationen von allen öffentlichen Typen, `structs`, ...

Code-Dateien (`.c`-Dateien):

- Liste der Funktionsprototypen aller privaten Funktionen.
- Konsistente Regeln für Reihenfolge der Funktionsdefinitionen. (Z.B. zuerst öffentlich (darunter alphabetisch), dann privat (darunter alphabetisch).)
- Länge von Funktionen höchstens etwa 100 Zeilen.

10.8 Sprachgebrauch

- Komplexe Ausdrücke sollten in kleiner Teile zerlegt werden (z.B. mit Hilfe von neuen Variablen.)
- Seltsame, ungewöhnliche, unverständliche und gefährliche Konstruktionen sollten vermieden werden.
- Tiefe Schachtelungen sollten vermieden werden.

- `if-else` statt `?:`.
- Keine Zuweisungen in Ausdrücken.
- Boolsche Ausdrücke sollten immer einen Vergleichsoperator enthalten.
- Vergleichsoperatoren sollten nur in Boolschen Ausdrücken enthalten sein.
- Immer explizite Typumwandlung statt automatischer Typumwandlung. Auch bei Argumenten von Funktionsaufrufen.
- Zählen sollte immer bei 0 beginnen (z.B. in `for`).
- Möglichst immer asymmetrische Grenzen verwenden ($0 \leq \mathrm{Index} < \mathrm{Länge/Größe/Anzahl}$).
- In `switch` vor jedem `case` (außer dem ersten) ein `break`.
- `break` und `continue` ansonsten möglichst selten verwenden.
- `goto` möglichst gar nicht verwenden.
- `return` vor dem Ende des Funktionsblocks nur in Ausnahmesituationen verwenden.
- Immer den Rückgabetyp deklarieren.
- Argumente (-Typen und -Namen) verschiedener Funktionen möglichst ähnlich wählen.
- Nicht zu viele Argumente.
- Nicht zu komplexe Ausdrücke bei Argumenten von Funktionsaufrufen.
- Keine `#define`s um die Syntax von C auszuhebeln.
- Vorsicht bei Makros: Kein `;` , Klammern von Makroparametern und dem ganzen Makro, bei Verwendung an Seiteneffekte denken.

10.9 Datengebrauch

- Gleitkommazahlen nicht auf Gleichheit vergleichen.
- `typedef` statt `#define` und `struct`-Typ.
- Möglichst wenig globale Daten.
- Möglichst keine tiefen Schachtelungen von Zeigern.
- Möglichst wenig `union`s.
- `sizeof` statt Konstante.
- *Magic Numbers* immer `#define`n. (*Magic Numbers* sind explizit angegebene konstante Zahlen außer 0 und 1, z.B. 10.)
- `static`-Variablen initialisieren.

Kapitel 11

Beliebte Fehler in C

(Dieses Kapitel ist zum Teil ein Auszug aus Andrew Koenig *Der C-Experte*, Addison-Wesley, 1989.)

11.1 Lexikalische Fallen

Ein C-Compiler betrachtet ein Programm nicht als eine Ansammlung einzelner Buchstaben, sondern – genauso wie der Mensch – liest er vielmehr die einzelnen Wörter. Die Wörter werden hier jedoch Tokens genannt. Unter Umständen können diese Tokens ähnlich oder sogar gleich aussehen und etwas komplett verschiedenes bedeuten.

11.1.1 `= != ==` (oder: = ist nicht ==)

In vielen Programmiersprachen (Pascal, Ada, Algol) wird `:=` für die Zuweisung und = für den Vergleich verwendet. In C hingegen wird = für die Zuweisung verwendet und == zum Vergleichen. Hier kommt einmal mehr die Schreibfaulheit der C-Entwickler zum Ausdruck: Die Zuweisung wird nämlich in der Regel viel häufiger als der Vergleich verwendet und somit spart man sich ein paar Buchstaben Tipparbeit. Ein sinnvoller Aspekt ist jedoch, dass C die Zuweisung als einen Operator behandelt, so dass man leicht Mehrfachzuweisungen zum Beispiel wie folgt schreiben kann: `a=b=c`

Dies führt jedoch auch zu Problemen. So ist es möglich, Zuweisungen zu schreiben, obwohl man eigentlich einen Vergleich durchführen wollte:

```
if (x = y)
   a = x;
```

Hier wird in Wirklichkeit zunächst der Wert von `x` mit dem von `y` überschrieben und danach getestet, ob dieser Wert ungleich 0 ist. Das Ergebnis ist also ein komplett anderes als bei einem Vergleich `x == y`. Einige Compiler geben daher an solchen Stellen eine Warnung aus, die meisten tun dies jedoch nicht.

Auch weniger gut sichtbare Fehler können auf diese Art entstehen. Will man zum Beispiel die Leerzeichen, Tabulatoren und Zeilenvorschübe in einer Datei überspringen, so schlägt der folgende Versuch fehl:

```
while (c = ' ' ||
       c == '\t' ||
       c == '\n' )
   c = getc(f);
```

Bei der gewählten Anordnung fällt das falsche = sofort ins Auge, aber schreibt man alles in eine Zeile, so ist das Auffinden eines solchen Fehlers schwieriger. Da der Operator = eine niedrigere Priorität hat als `||` wird der Variablen `c` der Wert des gesamten Ausdrucks

```
' ' || c == '\t' || c == '\n'
```

zugewiesen. Da `' '` immer ungleich 0 ist wird die Abfrage immer erfüllt. Ist das Einlesen von Zeichen auch nach Erreichen des Dateiendes erlaubt, so wird diese Schleife ewig laufen.

Sollte bei Bedingungen einmal dennoch eine Zuweisung erfolgen, so kann man dies wie folgt umschreiben: Anstelle von

```
if (x = y)
   ...
```

schreibt man einfach

```
if ((x = y) != 0)
   ...
```

Dies verhindert eventuell auftretende Warnungen und verdeutlicht die Absicht des Programmierers.

11.1.2 & und | ist nicht && oder ||

Auch bei diesen Operatoren treten leicht Verwechslungen auf: & bzw. | stellen eine bitweise Verknüpfung dar, wohingegen && bzw. || logische Verknüpfungsoperatoren sind.

11.1.3 Habgierige Analyse

Wer schon einmal mit regulären Ausdrücken zu tun hatte weiß, dass diese immer den längstmöglichen String ermitteln, welcher dem Suchmuster entspricht. Genauso ist ein C-Compiler: Er versucht immer zuerst das größtmögliche Token zu verwenden, die lexikalische Analyse ist also sozusagen habgierig. Dies führt zur Verwirrung bei einigen Tokens:

```
y = x/*p  /* p ist Pointer */;
```

Der Programmierer wollte hier offensichtlich eine Division durch den Wert auf den der Pointer `p` zeigt durchführen. Wegen der Habgier wird `/*` jedoch als Einleitung eines Kommentars interpretiert. Für den Compiler sieht diese Anweisung daher wie `y = x` aus! Leerzeichen oder Klammern können in solchen Situationen Abhilfe schaffen:

```
y = x/ *p  /* p ist Pointer */;
y = x/(*p)  /* p ist Pointer */;
```

11.1.4 Integerkonstanten

Ist das erste Zeichen einer Integerkonstante `0`, so wird diese als Oktalzahl interpretiert. `010` ist somit `8` und nicht, wie man vielleicht erwartet hätte `10`. Bei Oktalzahlen sind zudem nur Ziffern zwischen 0 und 7 erlaubt!

Hexadezimale Zahlen werden durch ein vorangestelltes `0x` eingeleitet und setzen sich aus den Ziffernsymbolen `0-9` und `A-F` (bzw. `a-f`) zusammen.

11.1.5 Strings und Zeichen

Ein Zeichen (`char`) wird in der Regel wie folgt definiert:

```
char c = 'a';
```

`'a'` wird intern in eine Integerzahl umgewandelt. Die Zuweisung ist also gleichbedeutend mit:

```
char c = 97;
```

oder

```
char c = 0141;
```

Einige Compiler lassen auch mehrere Zeichen zwischen den einfachen Hochkommas zu, da eine Integerzahl mehrere Bytes lang ist. Dies entspricht jedoch nicht ANSI-C, der tatsächliche Wert von `'no'` ist daher auch nicht exakt definiert.

Im Gegensatz zu den Zeichenkonstanten stellen die Stringkonstanten eine abgekürzte Schreibweise für einen Zeiger auf das erste Zeichen eines namenlosen Arrays dar. Dieses Array wird am Ende um ein Zeichen mit dem Wert `0` erweitert, welches das Ende des Strings markiert. Die Anweisung

```
printf("Hallo Welt\n");
```

ist also gleichbedeutend mit

```
char hallo[]={'H','a','l','l','o',
      ' ','W','e','l','t','\n',0);
printf(hallo);
```

Einige Compiler überprüfen nicht den Typ von Funktionsargumenten, so dass die folgende, falsche Anweisung zur Laufzeit ein interessantes Wirrwar von Zeichen auf den Bildschirm zaubert.

```
printf('\n');
```

11.2 Syntaktische Fallstricke

Es reicht leider nicht, wenn man nur die Tokens eines C-Programms versteht. Man muss auch wissen, wie diesen Tokens zu Deklarationen, Ausdrücken, Anweisungen und Programmen kombiniert werden. Diese Kombinationen sind in der Regel klar festgelegt, ihre Definition ist unter Umständen jedoch verwirrend. Dieser Abschnitt behandelt einige dieser undurchsichtigen syntaktischen Konstruktionen.

11.2.1 Funktionsdeklarationen

Um die Deklaration von zunächst kompliziert aussehenden Funktionsdeklarationen wie

```
(*(void(*)())0)();
```

zu verstehen, beherzigt man am besten die folgende Regel: *Deklarieren Sie die Funktion so, wie Sie sie verwenden*.

Um dies zu verstehen, betrachtet man zunächst die Variablendeklaration in C. Sie besteht aus zwei Teilen: einem Typ und den Deklaratoren. Für zwei Fließkommazahlen sieht das dann zum Beispiel so aus:

```
float f, g;
```

Da ein Deklarator wie ein Ausdruck behandelt wird kann man beliebig viele Klammern setzen und muss dies sogar in einigen Fällen, wie man gleich sehen wird und was auch in Abschnitt 11.2.2 entsprechend begründet wird.

```
float ((f));
```

bedeutet daher, dass `((f))` ein `float` ist und deshalb darauf geschlossen werden kann, dass `f` ebenfalls ein `float` ist.

Eine ähnliche Logik wird bei Funktionen und Zeigertypen angewendet:

```
float ff();
```

heißt, dass `ff()` ein `float` ist und daher muss `ff` eine Funktion sein. Ebenso gilt bei Pointern

```
float *pf;
```

dass der Ausdruck `*pf` einen `float` ergibt und somit `pf` ein Zeiger auf einen `float` repräsentiert.

Diese Formen werden in Deklarationen genauso miteinander kombiniert wie bei Ausdrücken:

```
float *g(), (*h)();
```

besagt also, dass `*g()` und `(*h)()` `float`-Ausdrücke sind. Da `()` gegenüber `*` den Vorrang hat (siehe Abschnitt 11.2.2), bedeutet `*g()` dasselbe wie `*(g())`. Bei `g` handelt es sich also um eine Funktion, die einen Zeiger auf einen `float` zurückliefert. `h` ist ein Zeiger auf eine Funktion, die einen `float` zurückgibt.

Will man den Datentyp einer Variablen angeben, so geschieht das ganz einfach, indem man den Variablennamen weglässt und das ganze in Klammern setzt. In obigem Beispiel ist `h` also vom Datentyp

```
(float (*)())
```

Wir können nun den Ausdruck `(*(void(*)())0)` – welcher ein Unterprogramm aufruft, das an der Adresse `0` gespeichert ist – in zwei Schritten analysieren. Zunächst gehen wir von einer Variablen `fp` aus, die einen Funktionszeiger enthält. Dabei wollen wir die Funktion aufrufen, auf die `fp` zeigt. Das läßt sich folgendermaßen durchführen:

```
(*fp)();
```

Wenn `fp` ein Zeiger auf eine Funktion ist, dann ist `*fp` selbst eine Funktion, so dass sie mit `(*fp)()` aufgerufen werden kann. ANSI-C erlaubt eine Abkürzung mit `fp()`, dies ist jedoch nur eine abgekürzte Schreibweise!

Die Klammern um `*fp` im Ausdruck `(*fp)` sind unbedingt notwendig, da der Funktionsoperator `()` stärker bindet als der unäre Verweis-Operator `*`.

Der Zeiger auf die Funktion soll auf die Adresse `0` zeigen. Leider funktioniert `(*0)();` nicht, da der Operator `*` als Operand eine Adresse erfordert. Deshalb müssen wir `0` in einen Datentyp umwandeln, der als "Adresse einer Funktion, die den Wert `0` liefert" beschrieben werden kann.

Wenn `fp` ein Zeiger auf eine Funktion ist, die den Datentyp `void` liefert, dann ergibt `(*fp)()` einen `void`-Wert und die Deklaration nimmt somit folgende Gestalt an:

```
void (*fp)();
```

Wenn wir wissen, wie eine Variable deklariert wird, so wissen wir auch, wie man eine Konstante in diesen Datentyp umwandelt: wir brauchen bloß den Variablennamen weglassen und dies als `cast`-Anweisung zwischen zwei Klammern schreiben. Wir wandeln also `0` in einen "Zeiger auf eine Funktion, die den Datentyp `void` zurückgibt" um

```
(void(*)())0
```

und können nun `fp` durch `(void(*)())0` ersetzen:

```
(*(void(*)())0)();
```

Der Strichpunkt am Ende wandelt den Ausdruck in eine Anweisung um und sorgt somit dafür, dass das Unterprogramm an der Adresse `0` aufgerufen wird.

11.2.2 Rangfolge bei Operatoren

Operatoren haben unterschiedliche Prioritäten, das heißt, einige haben einen höheren Rang als andere und werden als erstes abgearbeitet. Um ein Beispiel aus dem ersten Abschnitt noch einmal aufzugreifen betrachten wir die folgende Bedingung:

```
if (x = y)
   ...
```

Diese hatten wir umgeschrieben in

```
if ((x = y) != 0)
   ...
```

Die Klammern um die Anweisung `x = y` sind notwendig, da `!=` Vorrang gegenüber dem Zuweisungsoperator `=` hat und somit als erstes interpretiert wird. Ohne die Klammern wäre der Ausdruck also gleichbedeutend mit

```
if (x = (y != 0))
   ...
```

Dies liefert aber nur wenn `y` gleich `1` ist, das genau gleiche Endergebnis wie `if (x = y) ...`

Die nachfolgende Tabelle gibt einen Überblick über alle Operatoren und die Rangfolge, nach der sie interpretiert werden:

Operator	Assoz.
`( ) [ ] -> .`	links
`! ~ ++ -- - (`*Typ*`) * & sizeof`	rechts
`* / %`	links
`+ -`	links
`<< >>`	links
`< <= > >=`	links
`== !=`	links
`&`	links
`^`	links
`|`	links
`&&`	links
`||`	links
`?:`	rechts
Zuweisungen	rechts
`,`	links

Die Operatoren, die am stärksten gebunden werden, sind diejenigen, die eigentlich gar keine Operatoren darstellen: Indizierung, Funktionsaufruf und Strukturauswahl. Diese sind alle links-assoziativ, das heißt, `a.b.c` bedeutet dasselbe wie `(a.b).c` und nicht `a.(b.c)`.

In der weiteren Reihenfolge kommen die einstelligen Operatoren, die binären, die arithmetischen und dann die Shift-Operatoren. Die relationalen, logischen und Zuweisungsoperatoren folgen, sowie der Bedingungsoperator. Das Schlußlicht bildet der Kommaoperator, er wird meist als Ersatz für den Strichpunkt verwendet, wenn ein Ausdruck anstelle einer Anweisung erforderlich ist, zum Beispiel in Makrodefinitionen und `for`-Schleifen-Initialisierung.

Hier noch einmal ein Beispiel für einen Operator mit rechter Assoziativität: der Zuweisungsoperator. Er wird von rechts nach links ausgewertet.

```
a = b = 0;
```

bedeutet somit dasselbe wie

```
b = 0;
a = b;
```

Ist man sich unsicher über die Rangfolge bzw. will man das Programm möglichst verständlich schreiben, so empfiehlt sich immer eine entsprechende Klammerung der Ausdrücke.

11.2.3 Kleiner `;` große Wirkung

Ein überflüssiger Strichpunkt kann harmlos sein. Im besten Fall ist es eine Null-Anweisung, die nichts bewirkt, im schlimmsten Fall steht er hinter einer `if`, `while` oder `for`-Anweisung, nach der genau eine Anweisung stehen darf. Das Ergebnis ist dann das gleiche, wie wenn diese Anweisungen überhaupt nicht da wären.

```
if (x > max);
   max = x;
```

bewirkt also nicht das gleiche wie das eigentlich gewünschte

```
if (x > max)
   max = x;
```

Auch das Vergessen eines Strichpunkts kann Fehler bewirken, welche der Compiler unter Umständen nicht beanstandet. So sieht zum Beispiel dieser Programmabschnitt

```
if (n < 3)
   return
x = n;
```

für den Compiler aus wie

```
if (n < 3)
   return (x = n);
```

Manche Compiler bemerken, dass diese Funktion nun ein Ergebnis vom Typ `int` zurückgibt, obwohl `void` erwartet wird. Ist der Rückgabewert einer Funktion jedoch nicht explizit angegeben, so wird in der Regel keine Warnung ausgegeben und der Rückgabetyp `int` angenommen. Dies zeigt auch, wie wichtig es ist, immer

den Rückgabetyp einer Funktion bei der Deklaration mit anzugeben. Für den Fall `n >= 3` ist es außerdem offensichtlich, dass das Codefragment etwas komplett anderes macht, als beabsichtigt war.

Auch bei der Deklaration von Strukturen kann ein vergessener Strichpunkt unerwartete Folgen haben:

```
struct logrec {
   int date;
   int time;
   int code;
}

main()
{
   ...
}
```

Zwischen dem ersten } und `main` fehlt ein Strichpunkt. Dies führt zu der Annahme, dass `main` den Datentyp `struct logrec` zurückgibt statt `int`.

11.2.4 `switch` immer mit `break`

Bei der `switch` Anweisung sind die `case`-Marken in C als echte Sprungmarken implementiert, so dass beim Erreichen des Endes eines `case`-Abschnittes das Programm ungehindert weiterläuft. Daher muss am Ende eines `case`-Abschnittes immer eine `break`-Anweisung folgen! Das folgende Fragment zeigt ein Beispiel für eine korrekte `switch`-Anweisung:

```
switch (farbe) {
   case 1: printf("rot");
           break;
   case 2: printf("gelb");
           break;
   case 3: printf("gruen");
           break;
}
```

Vergißt man hier jedoch die `break`'s, und nimmt man an, `farbe` hätte nun den Wert 2, so wird man als Ergebnis die Ausgabe `gelbgruen` erhalten.

Es gibt jedoch auch durchaus Fälle, in denen es sinnvoll ist, das `break` wegzulassen. Will man zum Beispiel beim Einlesen einer Datei Leerräume überspringen, so könnte man das so realisieren:

```
switch (...) {
   case '\n': zeilenzahl++;
   case '\t':
   case ' ':
      ...
}
```

Bei Zeilenvorschüben wird hier noch zusätzlich der Zähler für die Zeilenanzahl inkrementiert.

11.2.5 Funktionsaufrufe

Bei Funktionen, die keine Argumente benötigen muss man dennoch eine leere Argumentenliste übergeben:

```
f();
```

Die Angabe von `f;` alleine bewirkt, dass die Adresse der Funktion ermittelt, aber nicht aufgerufen wird, was keinen Sinn macht.

11.2.6 Wohin gehört das `else` ?

Wie bei vielen Programmiersprachen, gibt es auch in C Probleme bei verschachtelten `if-else`-Konstrukten:

```
if (x == 0)
   if (y == 0)
      printf("Ursprung\n");
else {
   z = x + y;
   f(&z);
}
```

Mit einem Editor, der die Einrückung für C vernünftig behandelt, hätte man den Fehler vermutlich sofort bemerkt, denn er hätte den Code so formatiert:

```
if (x == 0)
   if (y == 0)
      printf("Ursprung\n");
   else {
      z = x + y;
      f(&z);
   }
```

C ordnet einen `else`-Teil nämlich immer dem nächsten, nicht abgeschlossenen `if` innerhalb desselben Blocks zu. Betrachtet man die ursprüngliche Formatierung aus dem ersten Codeabschnitt, so wollte der Programmierer wohl zwei Hauptfälle unterscheiden: `x=0` und `x`$\neq$`0`. Dies kann man durch eine entsprechende Klammerung erreichen.

```
if (x == 0) {
   if (y == 0)
      printf("Ursprung\n");
}
else {
   z = x + y;
   f(&z);
}
```

Meist ist es besser ein paar Klammern zuviel zu setzen, als sich darauf zu verlassen, dass man (insbesondere bei einzeiligen `if`-Bodies) die Verschachtelungen richtig beachtet hat.

11.3 Semantische Fallstricke

Ein Satz kann von der Rechtschreibung her perfekt sein und eine tadellose Grammatik aufweisen, aber er kann trotzdem einen zweideutigen und unbeabsichtigten Inhalt haben. Dieses Kapitel beschäftigt sich mit solchen Problemen in C-Programmen

11.3.1 Zeiger und Arrays

Die Notation von Zeigern und Arrays ist in C untrennbar miteinander verbunden. Zwei Punkte sind hierbei besonders wichtig:

1. C hat nur eindimensionale Arrays, deren Größe mit einer Konstante zur Compilationszeit festgelegt sein muss. Das Element eines Arrays kann jedoch ein Objekt mit beliebigem Typ sein, darunter auch ein anderes Array, so dass es möglich ist, mehrdimensionale Arrays zu simulieren.

2. Mit einem Array kann man nur zwei Dinge anstellen: die Größe bestimmen und den Zeiger auf das Element `0` des Arrays ermitteln. Alle anderen Operationen werden in Wahrheit nur mit Zeigern durchgeführt.

Deklariert wird ein eindimensionales Array wie folgt:

```
int a[3];
```

Es sind auch Arrays von Strukturen möglich:

```
struct {
   int p[4];
   double x;
} b[17];
```

Ein zweidimensionales Array lässt sich oberflächlich sehr leicht deklarieren:

```
int kalender[12][31];
```

Dies besagt, dass `kalender` ein Array von 12 Arrays mit jeweils 31 `int`-Elementen ist (und kein Array von 31 Arrays mit jeweils 12 `int`-Elementen !). Alle Operationen auf `kalender` werden in der Regel intern in Pointeroperationen umgewandelt, mit Ausnahme des `sizeof`-Operators, der die Größe des gesamten Arrays liefert, so wie man das auch erwartet, nämlich 31·12·`sizeof(int)`.

Will man auf das i-te Element von `a` zugreifen, so kann man das so machen:

```
elem = a[i];
```

Die Arrayvariable `a` zeigt – wie im 2. Punkt angegeben – auf das erste Element `a[0]` des Arrays. `a` liefert also das gleiche wie `&a[0]`. Intern wird eine Indizierung dementsprechend umgeschrieben:

```
a[i]  →  *(a + i)
```

Dies ist natürlich gleichbedeutend mit `*(i + a)`, so dass man statt `a[i]` auch `i[a]` schreiben kann, wovon man aber besser Abstand nehmen sollte.

Bei zweidimensionalen Arrays zeigt

```
p = kalender[4];
```

auf das Element `0` des Arrays `kalender[4]`.

Auf den “Integer des 8. Mai’s” kann man daher auf verschiedene Arten zugreifen:

```
i = kalender[4][7];
i = *(kalender[4] + 7);
i = *(*(kalender + 4) + 7);
```

11.3.2 Zeiger sind keine Arrays

Gehen wir davon aus, wir haben zwei Strings `s` und `t`, welche wir zu einem String zusammensetzen wollen. Hierfür stehen uns die Bibliotheksfunktionen `strcpy` und `strcat` zur Verfügung. Eine einfacher Ansatz ist

```
char *r;
strcpy(r, s);
strcat(r, t);
```

welcher jedoch falsch ist und meist zum Absturz führt. Der Grund dafür ist, dass `r` in undefiniertes Gebiet zeigt und außerdem an dieser Stelle auch kein Speicherplatz reserviert ist. Die Lösung könnte sein, die Deklaration `char *r` zu ersetzen durch `char r[100];`. Dies

setzt aber voraus, dass die Gesamtlänge von `s` und `t` immer unter 99 Zeichen sein muss (1 Zeichen wird für die Endekennung `'\0'` benötigt).

Die beste Lösung ist, sich mittels `malloc` entsprechend viel Speicher vorher zu reservieren:

```
char *r, malloc();
r = (char *)malloc(
             strlen(s)
            +strlen(t)
            +1);
if (!r) {
   fehlermeldung();
   exit(1);
}
strcpy(r, s);
strcat(r, t);
...
free(r);
```

11.3.3 Array-Deklarationen als Parameter

Es gibt in C keine Möglichkeit, ein Array an eine Funktion zu übergeben. Stattdessen wird immer nur der Zeiger auf das Anfangselement des Zeigers übergeben

```
int strlen(char s[])
{
   ...
}
```

ist also identisch mit

```
int strlen(char *s)
{
   ...
}
```

Auch bei den beiden Argumenten von `main` sieht man häufig zwei verschiedene Implementationen:

```
main(int argc, char *argv[])
{
   ...
}
```

beziehungsweise

```
main(int argc, char **argv)
{
   ...
}
```

Diese Regel gilt jedoch nicht immer, wie das folgende zeigt:

```
extern char *s;
```

ist nicht das selbe wie

```
extern char s[];
```

11.3.4 Die Synechdoche

Synechdoche = *(Oxford English Dictionary)* "Ein umfassenderer Begriff für einen weniger umfassenden oder umgekehrt; ein Ganzes für einen Teil oder ein Teil als Ganzes; eine Gattung für die Spezies oder eine Spezies als Gattung".

Dies soll den häufigen Fehler beschreiben, dass ein Zeiger mit den adressierten Daten verwechselt wird:

```
char *p, *q;
p = "xyz";
q = p;
```

Mit den beiden letzen Anweisungen wird lediglich der Zeiger auf das Array `'x'`, `'y'`, `'z'`, `'\0'` in `p` bzw. `q` kopiert und nicht das gesamte Array.

11.3.5 `NULL`-Zeiger sind keine leeren Strings

Als einzige Integerkonstante wird der Wert 0 bei Bedarf automatisch in einen entsprechenden Zeiger umgewandelt, der sich von jedem zulässigen Zeiger unterscheidet. Er wird daher oft symbolisch wie folgt definiert:

```
#define NULL 0
```

Dieser NULL-Zeiger darf niemals dereferenziert werden, wie nachfolgend zu sehen: `p` sei ein NULL-Zeiger, das Ergebnis von

```
printf("%s",p);
```

ist dann nicht definiert und führt in der Regel zum Programmabsturz.

11.3.6 Zähler und asymmetrische Grenzen

Ein C-Array mit `n` Elementen hat kein Element mit dem Index `n` ! Die Elemente werden vielmehr von `0` bis `n-1` numeriert. Daher ist auch dieses Programm zum Initialisieren des Arrays `a` falsch:

```
int i, a[10];
for (i = 0; i <= 10; i++)
   a[i] = 0;
```

Um solche Fehler zu vermeiden definiert man am besten den Wertebereich eines Problems immer asymetrisch:

```
0 <= i < 10
```

Die obere Grenze der Indizes entspricht dann der Größe des Arrays. Um dies zu verdeutlichen sollte man daher das obige Programm immer wie folgt korrigieren:

```
int i, a[10];
for (i = 0; i < 10; i++)
   a[i] = 0;
```

und niemals

```
int i, a[10];
for (i = 0; i <= 9; i++)
   a[i] = 0;
```

11.3.7 Die Reihenfolge der Auswertung

Im Gegensatz zu der in Abschnitt 11.2.2 vorgestellten Rangfolge für Operatoren, welche festlegt, dass zum Beispiel `a + b * c` als `a + (b * c)` interpretiert wird, garantiert die Reihenfolge der Auswertung, dass es in

```
if (n != 0 && z / n < 5.0)
   ...
```

nicht zu einem Fehler "Division durch Null" kommt. Diese Rangfolge ist für die vier C-Operatoren `&&`, `||`, `?:` und `,` genau festgelegt, für die anderen ist sie compilerabhängig.

`&&` und `||` werten zuerst den linken Operanden aus und dann den rechten, falls dies noch erforderlich ist.

Der Operator `?:` nimmt drei Operanden `a ? b : c`, wertet zuerst `a` aus und dann entweder `b` oder `c`, je nachdem, welchen Wert `a` hatte.

Der Komma-Operator `,` wertet seinen linken Operanden aus, verwirft das Ergebnis und wertet dann den rechten aus. Achtung: der Komma-Operator hat nichts mit dem Komma in Funktionsaufrufen zu tun, bei denen die Reihenfolge der Argumentübergabe nicht definiert ist. So bedeutet auch `f(x, y)` etwas anderes als `g((x, y))`. Im letzten Beispiel erhält `g` nur ein Argument.

11.3.8 Die Operatoren `&&`, `||` und `!`

C hat zwei Klasse von logischen Operatoren, die nur in Sonderfällen ausgetauscht werden können: die Bit-Operatoren `&`, `|` und `~`, sowie die logischen Operatoren `&&`, `||` und `!`.

Die Bit-Operatoren behandeln ihre Operanden – wie der Name schon andeutet – bitweise. Zum Beispiel ergibt `10 & 12` den Wert `8` (= `1000` binär), da hier die Binärdarstellungen von `10` (= `1010`) und `12` (= `1100`) "ver-und-et" werden.

Im Gegensatz dazu verarbeiten die logischen Operatoren ihre Argumente so, als ob sie entweder "falsch" (gleich `0`) oder "wahr" (ungleich `0`) sind. Sie liefern als Ergebnis `1` für "wahr" bzw. `0` für falsch zurück. So liefert also `10 && 12` den Wert `1` und nicht `8` wie der Bit-Operator `&`. Außerdem werten die Operatoren `&&` und `||` unter Umständen den zweiten Operanden erst gar nicht aus, wie schon im vorangegangenen Abschnitt 11.3.7 erklärt wurde.

11.3.9 Integerüberlauf

C hat zwei Arten von Integerarithmetik: vorzeichenbehaftet (`signed`) und vorzeichenlos (`unsigned`). Bei der letzteren gibt es keinen Überlauf, alle vorzeichenlosen Operationen werden modulo 2^n durchgeführt, wobei n die Anzahl der Bits im Resultat angibt. Bei gemischten Operanden werden zuerst alle Operanden in vorzeichenlose umgewandelt, so dass auch hier kein Überlauf auftritt.

Ein Überlauf kann jedoch auftreten, wenn beide Operanden vorzeichenbehaftet sind. Das Ergebnis eines Überlaufs ist dann nicht definiert! Daher sollte man gegebenenfalls vor der Operation auf die Möglichkeit eines Überlaufs testen. Ein plumper Versuch wäre (`a` und `b` seien zwei nicht-negative `int`-Variablen) :

```
if (a + b < 0)
   fehler();
```

Dies funktioniert natürlich nicht. Sobald `a+b` einen Überlauf erzeugt hat, weiß man nicht, wie das Ergebnis lautet. Eine korrekte Methode wäre, die Operanden vor der Abfrage in vorzeichenlose `int`-Werte umzuwandeln:

```
if ((unsigned)a + (unsigned)b
     > INT_MAX)
   fehler();
```

INT_MAX repräsentiert den größten darstellbaren Integerwert und ist in der Include-Datei limits.h definiert. Die elegantere Methode ist jedoch diese:

```
if ( a > INT_MAX - b )
   fehler();
```

11.3.10 Die Rückgabe eines Ergebnisses von main

Das einfachste mögliche C-Programm

```
main() {}
```

enthält einen kleinen Fehler und wird von strikten Compilern auch mit einer Warnung bedacht. Von main wird nämlich erwartet, dass es einen int-Wert zurückgibt. Wird kein Wert zurückgegeben, so ist dies meist harmlos, es wird dann in der Regel ein unbestimmter Integerwert zurückgegeben und solange man diesen Wert nicht verwendet spielt es keine Rolle. Jedoch ist es so, dass das Betriebssystem anhand des Rückgabewertes entscheidet, ob das C-Programm fehlerfrei ausgeführt wurde. Es kann somit zu überraschenden Ergebnissen kommen, wenn ein anderes Programm sich auf diesen Fehlerstatus verläßt. Ein korrektes C-Programm sollte also zumindest so aussehen:

```
main()
{
   return 0;  /*oder: exit(0);*/
}
```

11.4 Der Präprozessor

Vor der eigentlichen Compilation kommt noch ein Vorverarbeitungsschritt: der Präprozessor. Präprozessorkommandos beginnen immer mit einem #. Der Präprozessor ermöglicht es, wichtige Programmabschnitte abzukürzen. So ermöglicht er unter anderem die globale Definition von konstanten Größen, welche dann auch entsprechend einfach und schnell anzupassen sind. Er ermöglicht es auch, einfache und geschwindigkeitskritische Funktionen zu ersetzen. Der Aufwand beim Aufruf einer Funktion ist nämlich teilweise nicht unerheblich, so dass man unter Umständen bestrebt sein wird, etwas zu haben, das so ähnlich aussieht wie eine Funktion, aber doch weniger Aufwand nach sich zieht. Zum Beispiel werden getchar und putchar normalerweise als Makros implementiert um zu vermeiden, dass bei jeder Zeicheneingabe oder -ausgabe eine Funktion aufgerufen werden muss.

Beim Einsatz von Makros sollte man eines nie vergessen: Sie ersetzen nur Text im Programm, ein Makro wird vorher nicht auf seine korrekte Syntax geprüft. Man sollte sich daher immer überlegen, wie das Ergebnis im Programm aussieht, nachdem das Makro eingepflanzt wurde.

11.4.1 Leerzeichen und Makros

Bei der Definition ist zu beachten, dass zwischen dem Makronamen und seinen Parametern – welche übrigens auch komplett wegfallen können, im Gegensatz zu Funktionen, wo zumindest eine Leere Menge () übergeben werden muss – keine Leerzeichen auftreten dürfen. Das folgende Beispiel verdeutlicht dies:

```
#define f (x) ((x) - 1)
```

Man weiß nun nicht, was dieser Ausdruck darstellt. Soll f(x) ersetzt werden durch ((x) - 1), oder soll f durch (x) ((x) - 1) ersetzt werden ? Da keine Leerzeichen zwischen Name und Parametern erlaubt sind, wird sich der Präprozessor für die zweite Möglichkeit entscheiden.

Für Makroaufrufe gilt diese Regel allerdings nicht, so dass f(3) und f (3) beide den Wert 2 liefern.

11.4.2 Makros sind keine Funktionen

Zunächsteinmal sollte man darauf achten, dass man in Makros immer alle verwendeten Argumente einklammert. Um dies zu verdeutlichen betrachten wir diese falsche Implementation der Absolutfunktion:

```
#define abs(x) x > 0 ? x : -x
```

Verwendet man im Programm nun den Ausdruck

```
abs(a - b)
```

so ergibt sich nach dem Durchlauf durch den Präprozessor

```
a - b > 0 ? a - b : -a - b
```

was für a - b ≤ 0 natürlich ein falsches Ergebnis liefert, denn dieses müsste -(a - b) lauten und nicht -a - b. Bei Verwendung von Makros in größeren Ausdrücken ergeben sich bei fehlender Klammerung zudem eventuell unerwünschte Effekte wegen der Einhaltung von Rangfolgen. Daher sollte das obige Makro wie folgt geschrieben werden:

```
#define abs(x) (((x)>0)?(x):-(x))
```

Auch wenn Makrofunktionen voll geklammert sind, kann es trotzdem zu Problemen kommen:

```
#define max(a,b) ((a)>(b)?(a):(b))
nmax = x[0];
i = 1;
while (i < n)
   nmax=max(nmax, x[i++]);
```

Wäre `max` als Funktion implementiert gäbe es keine Probleme, als Makro kann es jedoch passieren, dass `x[i++]` zweimal ausgewertet wird, nämlich immer dann, wenn `nmax` kleiner ist. `i` wird somit insgesamt um `2` erhöht, statt nur – wie gewünscht – um `1`. Solche doppelten Aufrufe in Makros sollte man – auch in Hinblick auf den Mehraufwand – daher bei der Makrodefinition vermeiden, oder wenn dies nicht möglich ist, so sollte man das Makro mit der entsprechenden Vorsicht aufrufen:

```
#define max(a,b) ((a)>(b)?(a):(b))
nmax = x[0];
for (i = 1; i < n; i++)
   nmax=max(nmax, x[i]);
```

11.4.3 Makros sind keine Anweisungen

Man ist verleitet, Makros so zu definieren, dass sie wie Anweisungen verwendet werden können, aber dies ist erstaunlich schwierig. Ein Beispiel ist das Makro `assert` (siehe: `man assert`). Dieses könnte man zum Beispiel versuchen wie folgt zu realisieren:

```
#define assert(e) if (!e)      \
  assert_err(__FILE__,__LINE__);
```

Setzt man dieses Makro jedoch innerhalb von `if`-Schleifen ein, so kann es zu den selben Effekten kommen, wie sie in Abschnitt 11.2.6 besprochen wurden. In diesem Fall wird das `assert`-Makro unerwünschte Auswirkungen auf den Programm-Code haben:

```
if (x > 0 && y > 0)
   assert (x > y);
else
   assert (y > x);
```

Leider hilft in diesem Fall auch eine entsprechende Klammerung innerhalb des Makros nichts, wie hier geschehen:

```
#define assert(e) { if (!e)     \
  assert_err(__FILE__,__LINE__);}
```

Setzt man dieses Makro in obiges Beispiel ein, so wird man leicht sehen, dass sich zwischen der Klammer des Makros } und dem `else` noch ein Strichpunkt befindet, was einen Syntaxfehler zur Folge hat. Eine richtige Methode, um `assert` zu definieren ist zum Beispiel, das ganze als Ausdruck umzuschreiben:

```
#define assert(e) ((void)((e)|| \
 _assert_err(__FILE__,__LINE__)));
```

Hierbei wird die in Abschnitt 11.3.7 erwähnte Eigenschaft des `||`-Operators ausgenutzt.

11.4.4 Makros sind keine Typdefinitionen

Sehr häufig werden Makros dazu verwendet, um den Datentyp von mehreren Variablen an einer Stelle zu definieren, so dass man diesen später bequem ändern kann:

```
#define FOOTYPE struct foo
FOOTYPE a;
FOOTYPE b, c;
```

Es ist jedoch besser, eine Typdefinition zu verwenden wie man an folgendem Beispiel leicht sieht:

```
#define T1 struct foo *
typedef struct foo *T2;

T1 a, b;
T2 c, d;
```

Obwohl die Definitionen von T1 und T2 auf den ersten Blick das gleiche bewirken, sieht man jedoch sofort den Unterschied, wenn man die erste Deklaration nach dem Durchlauf durch den Präprozessor betrachtet:

```
struct foo *a, b;
```

`b` wird also nicht wie beabsichtigt als Zeiger auf eine Struktur definiert.

11.5 Tips && Tricks

Man sollte niemals ein Programm schnell hinschreiben (am besten noch kurz vor dem Schlafengehen) um dann etwas zu erhalten, das scheinbar funktioniert. Wenn es dann nämlich einmal erwartungsgemäß nicht das tut,

was es sollte, so wird es meist schwierig, den versteckten Fehler aufzuspüren. Deshalb sollte man immer gründlich nachdenken, was man haben möchte und wie man es entsprechend vollständig lösen kann, bevor man beginnt wilden Code zu produzieren.

Es sollen nun ein paar Tips vorgestellt werden, mit deren Hilfe man einige typischen Fehler, wenn nicht vermeiden, so doch zumindest minimieren kann.

11.5.1 Ratschläge

Absichten klarmachen

Bei Vergleichen mit Konstanten sollte man diese immer auf die linke Seite schreiben. So hätte der Fehler aus Abschnitt 11.1.1 sofort zu einer Fehlermeldung geführt, da Konstanten kein Wert zugewiesen werden kann:

```
while (' '  = c ||   /* Fehler! */
       '\t' == c ||
       '\n' == c )
   c = getc(f);
```

Außerdem sollte man lieber zuviele Klammern setzen, als zuwenige. Auch dies kann zu einer besseren Erkennung von unbeabsichtigten Fehlern durch den Compiler führen.

Auch einfache Fälle testen

Viele Programme stürzen ab, wenn sie eine leere Eingabe oder nur ein Element erhalten. Daher sollten auch immer die einfachen Fälle berücksichtigt und überprüft werden.

Asymmetrische Grenzen

Man muss beachten, dass Arrays in C schon bei `0` beginnen und somit auch ein Element früher enden, als man dies vielleicht vielleicht erwartet hätte. Siehe dazu auch Abschnitt 11.3.6.

Bugs verstecken sich in dunklen Ecken

Man sollte sich an die gut bekannten Konstrukte der Sprache halten und nicht ein spezielles Feature eines bestimmten Compilers unbedingt verwenden wollen. Dies kann bei anderen Compilern oder auf anderen Maschinen zu – eventuell nicht sofort sichtbaren – Fehlern führen. Man sollte sich auch nicht ohne weiteres darauf verlassen, dass Bibliotheksfunktionen sauber implementiert sind. Diese Tips gelten im übrigen auch uneingeschränkt für OpenGL.

Defensiv programmieren

Ein Programm sollte *alle* Fälle abdecken, die auftreten können. Ereignisse die nicht erwartet werden, aber passieren können treten meist viel zu häufig ein. Ein robustes Programm deckt alle Fälle ab und behandelt diese auch entsprechend.

Kapitel 12

Ausblick auf dreidimensionales OpenGL

Die Programmierung von dreidimensionalen Graphiken ist im Vergleich zu zweidimensionalen Graphiken meistens deutlich aufwendiger; vor allem wegen der (eventuell perspektivischen) Projektion auf einen zweidimensionalen Bildschirm, der Berechnung von Beleuchtungseffekten und der komplizierteren geometrischen Transformationen in drei Dimensionen. Hier ein kleines Beispiel für eine dreidimensionale Graphik in OpenGL:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <GL/glut.h>

void malen(void);

int main(int argc, char **argv)
{
   GLfloat light0_pos[] =
      {100., 100., 100., 1.0};
   GLfloat light0_color[] =
      {1., 0., 1., 1.0};
   GLfloat ambient_light[] =
     {0.5, 0.5, 0.5, 1.0};

   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_RGB |
      GLUT_DEPTH);
   glutInitWindowSize(400, 300);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Hallo!");
   glClearColor(0.0, 0.0, 0.0, 0.0);

   glLightModelfv(
      GL_LIGHT_MODEL_AMBIENT,
      ambient_light);
   glLightfv(GL_LIGHT0, GL_POSITION,
     light0_pos);
   glLightfv(GL_LIGHT0, GL_DIFFUSE,
     light0_color);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);

   glEnable(GL_DEPTH_TEST);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glFrustum(-2., 2., -1.5, 1.5,
      1., 20.);
   glutDisplayFunc(&malen);
   glutMainLoop();
   return(0);
}

void malen(void)
{
   glClear(GL_COLOR_BUFFER_BIT |
      GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0., 0., -5.);
   glRotatef(-30., 0.0, 1.0, 0.0);
   glutSolidTeapot(2.);
   glFlush();
}
```

Eine wichtige Änderung ist der Depth-Buffer, mit dem Verdeckungsberechnungen sehr effizient durchgeführt werden können. Mit

```
   glutInitDisplayMode(GLUT_RGB |
      GLUT_DEPTH);
```

wird ein Depth-Buffer angefordert, und mit

```
glEnable(GL_DEPTH_TEST);
```

für Verdeckungstests aktiviert. In `malen()` muss nun auch der Depth-Buffer gelöscht werden:

```
glClear(GL_COLOR_BUFFER_BIT |
   GL_DEPTH_BUFFER_BIT);
```

Beleuchtungsberechnungen werden mit

```
glEnable(GL_LIGHTING);
```

aktiviert. Für die Beleuchtung wird zum einen mit

```
glLightModelfv(
   GL_LIGHT_MODEL_AMBIENT,
   ambient_light);
```

ein allgemeines (ambientes) Licht eingeschaltet und zum anderen mit

```
glLightfv(GL_LIGHT0, GL_POSITION,
  light0_pos);
glLightfv(GL_LIGHT0, GL_DIFFUSE,
  light0_color);
glEnable(GL_LIGHT0);
```

eine Lichtquelle an die Position `light0_pos` gesetzt, die in der Farbe `light0_color` scheint.

Die perspektivische Projektion wird mit

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-2., 2., -1.5, 1.5,
   1., 20.);
```

definiert. Damit wird ein Sichtfeld definiert, das in z-Richtung von -1 bis -20 reicht, und (in der $z = -1$-Ebene) in x-Richtung von -2 bis $+2$ und in y-Richtung von -1.5 bis 1.5. Wichtig ist dabei vor allem, dass die Standard-Blickrichtung in OpenGL die negative z-Achse ist.

In der Funktion `malen()` wird schließlich mit der Funktion

```
glutSolidTeapot(2.);
```

eine Teekanne aus OpenGL-Polygonen gemalt, deren Größe mit einem Parameter eingestellt werden kann. Diese Teekanne wird mit

```
glRotatef(-30., 0., 1., 0.);
```

um 30 Grad um die y-Achse rotiert und mit

```
glTranslatef(0., 0., -5.);
```

entlang der negativen z-Achse verschoben.

Wem dieser Ausblick nicht reicht, der findet im „OpenGL Programming Guide“ (siehe Literaturliste) noch wesentlich mehr Details zur OpenGL-Programmierung.

Übungen zum fünften Tag

In den folgenden Aufgabe soll mit Hilfe der von GLUT zur Verfügung gestellten Funktionalitäten eine kleine interaktive 3D-Anwendung realisiert werden.Die Aufgaben bauen dabei auf dem Beispielprogramm aus Kapitel 12 auf.

Aufgabe V.1

Erweitern Sie das Beispielprogramm aus Kapitel 12 um eine einfache Maussteuerung, mit der die drei Rotationswinkel ϕ_x. ϕ_y und ϕ_z um die drei Koordinatenachsen variiert werden können. Die drei Rotationen sollen immer in der folgenden Reihenfolge durchgeführt werden: zuerst um ϕ_y um die y-Achse, dann um ϕ_x um die x-Achse und schließlich um ϕ_z um die z-Achse. Im Folgenden ist genauer erläutert, wie die Mausbewegung umgesetzt werden soll:

Bei Betätigung der linken Maustaste soll die Mausbewegung in x-Richtung in eine Änderung von ϕ_y und die in y-Richtung in eine Änderung von ϕ_x umgesetzt werden. Bei zusätzlicher Betätigung der SHIFT-Taste soll nur die Mausbewegung in x-Richtung, bei Betätigung der ALT-Taste nur die Mausbewegung in y-Richtung berücksichtigt werden. Bei Betätigung der mittleren Maustaste soll die Mausbewegung in x-Richtung in eine Änderung von ϕ_z umgesetzt werden. Die Bewegung in y-Richtung wird ignoriert.

Ausgabe V.2

In dieser Aufgabe soll der KeyboardFunc-Callback implementiert werden um verschiedene Einstellungen über Tastendruck zu ermöglichen. Realisiern Sie folgende Funktionalitäten:

1. Über eine Taste soll zwischen solider und Gitternetz-Darstellung des Teekessels gewechselt werden können.

2. Eine weitere Taste soll das Ein- bzw. Ausschalten der Beleuchtung ermöglichen.

3. Mit einer weiteren Taste, soll die Position des Objekts auf den Ausgangzustand zurückgesetzt werden.

4. Durch gleichzeitiges Betätigen von CTRL und Q soll das Programm beendet werden.

5. Über eine geeignete Taste (z.B. `h`) soll auf der Standardausgabe eine Hilfe zur obigen Tastenbelegung ausgegeben werden.

Ausgabe V.3

Erweitern Sie ihr Programm um die Möglichkeit einher Animation des Teekessels, bei der der Teekessel fortlaufend um die y-Achse rotiert wird. Die Animation soll dabei über eine geeignete Taste gestartet bzw. gestoppt werden können. Zusätzlich soll über die Pfeiltasten `UP` und `DOWN`, die Rotationsgeschwindigkeit erhöht bzw. gesenkt werden können. Außerdem soll mit einer weiteren Taste die Rotationsrichtung geändert werden können. Nehmen Sie diese Animationssteuerung in die Hilfebeschreibung aus Aufgabe V.2 auf.

Ausgabe V.3

Fügen Sie zu Ihrem Programm ein Menü hinzu. Dieses Menü soll für jedes geometrische Objekt, dass von der GLUT-Bibliothek zur Verfügung gestellt wird, also Teekessel, Kugel, usw. (s. Kapitel 8.7), einen Eintrag enthalten. Beim Anklicken eines Menüeintrags soll die Darstellung des bisherigen Objektes durch die des neu ausgewählten ersetzt werden. Die sonstigen Einstellmöglichkeiten aus den vorangegangenen Aufgaben sollen für alle auswählbaren Objekte weiterhin funktionieren.

Fügen Sie außerdem in einer Ecke des OpenGL-Fensters eine Textausgabe des aktuellen Systemzustands ein. Also z.B. welches Objekt dargestellt wird, ob die Beleuchtung eingeschaltet ist, ob die Animation läuft, ...

Literatur

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1977.
- Andrew Koenig, *C Traps and Pitfalls*, Addison-Wesley, 1989.
- David Straker, *C Style : Standards and Guidelines*, Prentice Hall, 1992.
- Mason Woo, Jackie Neider, Tom David, Dave Shriner, Tom Davis, *OpenGL 1.2 Programming Guide*, 3rd ed., Addison Wesley, 1999.