

# **Hardwarenahe Programmierung**

Wintersemester 2018/19  
Prof. Dr. rer. nat. Peter Gerwinski

Stand: 2. Dezember 2018

Soweit nicht anders angegeben:

Text und Bilder: Copyright © 2012, 2013, 2015, 2016, 2018 Peter Gerwinski

Lizenz: CC-by-sa (Version 3.0) oder GNU GPL (Version 3 oder höher)

Sie können dieses Skript einschließlich Vortragsfolien, Beispielprogramme und sonstiger Lehrmaterialien unter <https://gitlab.cvh-server.de/pgerwinski/hp> herunterladen.

# Inhaltsverzeichnis

<b>1 Einführung</b>	<b>5</b>
1.1 Was ist hardwarenahe Programmierung?	5
1.2 Programmierung in C	5
<b>2 Einführung in C</b>	<b>6</b>
2.1 Hello, world!	6
2.2 Programme compilieren und ausführen	7
2.3 Elementare Aus- und Eingabe	8
2.4 Elementares Rechnen	11
2.5 Verzweigungen	12
2.6 Schleifen	15
2.7 Seiteneffekte	17
2.8 Strukturierte Programmierung	19
2.9 Funktionen	20
2.10 Zeiger	22
2.11 Arrays und Strings	22
2.11.1 Arrays	22
2.11.2 Strings	23
2.12 String-Operationen	25
2.13 Parameter des Hauptprogramms	27
2.14 Strukturen	28
2.15 Dateien und Fehlerbehandlung	34
<b>3 Bibliotheken</b>	<b>36</b>
3.1 Der Präprozessor	36
3.2 Bibliotheken einbinden	37
3.3 Bibliotheken verwenden (Beispiel: OpenGL)	39
3.4 Projekt organisieren: make	42
3.4.1 make-Regeln	42
3.4.2 make-Macros	43
3.4.3 Fazit: 3 Sprachen	43
<b>4 Hardwarenahe Programmierung</b>	<b>44</b>
4.1 Bit-Operationen	44
4.1.1 Zahlensysteme	44
4.1.2 Bit-Operationen in C	46
4.2 Programmierung von Mikrocontrollern	47
4.3 I/O-Ports	47
4.4 Interrupts	49
4.5 volatile-Variable	50
4.6 Byte-Reihenfolge – Endianness	50

4.6.1	Konzept	50
4.6.2	Dateiformate	51
4.6.3	Datenübertragung	51
4.7	Binärdarstellung von Zahlen	52
4.8	Speicherausrichtung – Alignment	53
<b>5</b>	<b>Algorithmen</b>	<b>54</b>
5.1	Differentialgleichungen	54
5.1.1	Beispiel: Pendelschwingung	55
5.1.2	Das explizite Euler-Verfahren	55
5.2	Rekursion	57
5.3	Aufwandsabschätzungen	58
5.3.1	Sortieralgorithmen	58
5.3.2	Landau-Symbole	58
<b>6</b>	<b>Objektorientierte Programmierung</b>	<b>59</b>
6.0	Dynamische Speicherverwaltung	59
6.1	Konzepte und Ziele	60
6.2	Beispiel: Zahlen und Buchstaben	60
6.3	Unions	62
6.4	Beispiel: graphische Benutzeroberfläche (GUI)	63
6.5	Virtuelle Methoden	63
6.6	Einführung in C++	65
<b>7</b>	<b>Datenstrukturen</b>	<b>65</b>
7.1	Stack und FIFO	65
7.2	Verkettete Listen	66
7.3	Bäume	66

# 1 Einführung

## 1.1 Was ist hardwarenahe Programmierung?

In der Programmierung steht „hardwarenah“ für maximale Kontrolle des Programmierers über das genaue Verhalten der Hardware. Im Gegensatz zur abstrakten Programmierung, in der man als Programmierer in einer für Menschen möglichst komfortablen Weise das gewünschte Verhalten des Computers beschreibt und den Programmierwerkzeugen überläßt, auf welche Weise genau der Computer dies umsetzt, geht es in der hardwarenahen Programmierung darum, das Verhalten des Prozessors und jeder einzelnen Speicherzelle genau zu kennen.

Abbildung 1: Wissenschaftliche Disziplinen mit Bezug zur Informatik, angeordnet nach Abstraktionsgrad ihres jeweiligen Gegenstandes

Im Gegensatz zu z. B. Lebewesen werden Computer von Menschen entwickelt und gebaut. Daher ist es grundsätzlich möglich, sie durch hardwarenahe Programmierung vollständig zu verstehen und zu beherrschen.

## 1.2 Programmierung in C

Ein großer Teil dieser Vorlesung wird der Programmierung in der Programmiersprache C gewidmet sein.

Warum C?

C hat sich als Kompromiß zwischen einer Hochsprache und maximaler Nähe zur Hardware sehr weit etabliert. Es läuft auf nahezu jeder Plattform (= Kombination aus Hardware und Betriebssystem) und stellt somit einen „kleinsten gemeinsamen Nenner der Programmierung“ dar. C orientiert sich sehr eng an der Funktionsweise der Computer-Hardware und wird daher auch als „High-Level-Assembler“ bezeichnet.

Wie die Assembler-Sprache, die Sie in *Grundlagen Rechnertechnik* kennenlernen werden, ist C ein Profi-Werkzeug und als solches „leistungsfähig, aber gefährlich“. Programme können in C sehr kompakt geschrieben werden. C kommt mit verhältnismäßig wenigen Sprachelementen aus, die je nach Kombination etwas anderes bewirken. Dies hat zur Folge, daß einfache Schreibfehler, die in anderen Programmiersprachen als Fehler bemängelt würden, in C häufig ein ebenfalls gültiges Programm ergeben, das sich aber völlig anders als beabsichtigt verhält.

C wurde gemeinsam mit dem Betriebssystem Unix entwickelt und hat mit diesem wichtige Eigenschaften gemeinsam:

- **Kompakte Schreibweise:** Häufig verwendete Konstrukte werden möglichst platzsparend notiert. Wie in C, kann auch unter Unix ein falsch geschriebenes Kommando ein ebenfalls gültiges Kommando mit anderer Wirkung bedeuten.
- **Baukastenprinzip:** In C wie in Unix bemüht man sich darum, den unveränderlichen Kern möglichst klein zu halten. Das meiste, was man in C tatsächlich benutzt, ist in Form von Bibliotheken modularisiert; das meiste, was man unter Unix tatsächlich benutzt, ist in Form von Programmen modularisiert.
- **Konsequente Regeln:** In C wie in Unix bemüht man sich darum, feste Regeln – mathematisch betrachtet – möglichst einfach zu halten und Ausnahmen zu vermeiden. (Beispiel: Unter MS-DOS und seinen Nachfolgern wird eine ausführbare Datei gefunden, wenn sie sich *entweder* im aktuellen Verzeichnis *oder* im Suchpfad befindet. Unter Unix wird sie gefunden, wenn sie sich im Suchpfad befindet. Es ist unter Unix möglich, das aktuelle Verzeichnis in den Suchpfad aufzunehmen; aus Sicherheitserwägungen heraus geschieht dies jedoch üblicherweise nicht.)
- **Kein „Fallschirm“:** C und Unix führen Befehle ohne Nachfrage aus. (Beispiel: Ein eingegebener Unix-Befehl zum Formatieren einer Festplatte wird ohne Rückfrage ausgeführt.)

Trotz dieser Warnungen besteht bei Programmierübungen in C normalerweise *keine* Gefahr für den Rechner. Moderne PC-Betriebssysteme überwachen die aufgerufenen Programme und beenden sie notfalls mit einer

Fehlermeldung („Schutzverletzung“). Experimente mit Mikrocontrollern, die im Rahmen dieser Lehrveranstaltung stattfinden werden, erfolgen ebenfalls in einer Testumgebung, in der kein Schaden entstehen kann.

Bitte nutzen Sie die Gelegenheit, in diesem Rahmen Ihre Programmierkenntnisse zu trainieren, damit Sie später in Ihrer beruflichen Praxis, wenn durch ein fehlerhaftes Programm ernsthafter Schaden entstehen kann, wissen, was Sie tun.

## 2 Einführung in C

### 2.1 Hello, world!

Das folgende Beispiel-Programm (Datei: [hello-1.c](#)) gibt den Text „Hello, world!“ auf dem Bildschirm aus:

```
#include <stdio.h>

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

Dieses traditionell erste – „einfachste“ – Beispiel enthält in C bereits viele Elemente, die erst zu einem späteren Zeitpunkt zufriedenstellend erklärt werden können:

- **#include <stdio.h>**

Wir deuten diese Zeile im Moment so, daß uns damit gewisse Standardfunktionen (darunter `printf()` – siehe unten) zur Verfügung gestellt werden.

Diese Betrachtungsweise ist nicht wirklich korrekt und wird in Abschnitt 3.1 genauer erklärt.

- **int main (void) { ... }**

Dies ist das C-Hauptprogramm. Das, was zwischen den geschweiften Klammern steht, wird ausgeführt.

Auch hier wird zu einem späteren Zeitpunkt (Abschnitt 2.9) genauer erklärt werden, was die einzelnen Elemente bedeuten und welchen Sinn sie haben.

Im folgenden soll nun der eigentliche Inhalt des Programms erklärt werden:

```
printf ("Hello,_world!\n");
return 0;
```

- Bei beiden Zeilen handelt es sich um sogenannte *Anweisungen*.
- Jede Anweisung wird mit einem Semikolon abgeschlossen.
- Bei `printf()` handelt es sich um einen *Funktionsaufruf*, dessen Wirkung darin besteht, daß der zwischen den Klammern angegebene *Parameter* (oder: das *Argument*) der Funktion auf dem Standardausgabegerät ausgegeben wird. (In unserem Fall handelt es sich dabei um einen Bildschirm.) Der Name „`printf`“ der Funktion steht für „print formatted“ – formatierte Ausgabe.
- `"Hello,_world!\n"` ist eine *Konstante* vom Typ *String* (= Zeichenkette).
- `\n` ist eine *Escape-Sequenz*. Sie steht für ein einzelnes, normalerweise unsichtbares Zeichen mit der Bedeutung „neue Zeile“.
- Die Anweisung `return 0` bedeutet: Beende die laufende Funktion (hier: `main()`, also das Hauptprogramm) mit dem Rückgabewert 0. (Bedeutung: „Programm erfolgreich ausgeführt.“ – siehe Abschnitt 2.9.)

## 2.2 Programme compilieren und ausführen

Der Programmtext wird mit Hilfe eines Eingabeprogramms, des *Texteditors*, in den Computer eingegeben und als Datei gespeichert. Als Dateiname sei hier `hello-1.c` angenommen. Die Dateiendung `.c` soll anzeigen, daß es sich um einen Programmquelltext in der Programmiersprache C handelt.

Die `.c`-Datei ist für den Computer nicht direkt ausführbar. Um eine ausführbare Datei zu erhalten, muß das Programm zuerst in die Maschinsprache des verwendeten Computers übersetzt werden. Diesen Vorgang nennt man *Compilieren*.

In einer Unix-Shell mit installierter GNU-Compiler-Collection (GCC; frühere Bedeutung der Abkürzung: GNU-C-Compiler) geschieht das Compilieren durch Eingabe der folgenden Zeile, der *Kommandozeile*:

```
$ gcc hello-1.c -o hello-1
```

Das Zeichen `$` steht für die *Eingabeaufforderung* (oder das *Prompt*) der Unix-Shell. Es kann auch anders aussehen, z. B. `pc42:~$` oder auch `cassini/home/peter/bo/2018ws/hp/script>`. Die Eingabeaufforderung wird vom Computer ausgegeben; die Kommandozeile rechts daneben müssen wir eingeben und mit der Eingabetaste (Enter) bestätigen.

`gcc` ist ein Befehl an den Computer, nämlich der Name eines Programms, das wir aufrufen wollen (hier: der Compiler). Die darauf folgenden Teile der Kommandozeile heißen die *Parameter* oder *Argumente* des Befehls.

Der Parameter `hello-1.c` ist der Name der Datei, die compiliert werden soll.

`-o` ist eine *Option* an den Compiler, mit der man ihm mitteilt, daß der nächste Parameter `hello-1` der Name der ausführbaren Datei ist, die erzeugt werden soll.

Unter Unix ist es üblich, ausführbaren Dateien *keine* Endung zu geben. Unter Microsoft Windows wäre es stattdessen üblich, die ausführbare Datei `hello-1.exe` zu nennen.

Um von einer Unix-Shell aus ein Programm aufzurufen, gibt man dessen vollständigen Namen – einschließlich Verzeichnispfad und eventueller Endung – als Kommando ein:

```
$ ./hello-1
```

Der Punkt steht für das aktuelle Verzeichnis; der Schrägstrich trennt das Verzeichnis vom eigentlichen Dateinamen.

Wenn sich ein Programm im Suchpfad befindet (z. B.: `gcc`), darf die Angabe des Verzeichnisses entfallen. (Den Suchpfad kann man sich mit dem Kommando `echo $PATH` anzeigen lassen.) Aus Sicherheitsgründen steht das aktuelle Verzeichnis unter Unix üblicherweise *nicht* im Suchpfad.

Dateiendungen dienen unter Unix nur der Übersicht, haben aber keine technischen Konsequenzen:

- Ob eine Datei als ausführbar betrachtet wird oder nicht, wird nicht anhand einer Endung, sondern über ein *Dateiattribut* entschieden. Die Dateiattribute werden beim Listen des Verzeichnisinhalts angezeigt:

```
$ ls -l
-rwxr-x--- 1 peter ainf 6294  4. Okt 14:34 hello-1
-rw-r--r-- 1 peter ainf   82  4. Okt 15:11 hello-1.c
```

Jedes `r` steht für „read“ (Datei lesbar), jedes `w` für „write“ (Datei schreibbar) und jedes `x` für „execute“ (Datei ausführbar). Von links nach rechts stehen die `rw`-Gruppen für den Besitzer der Datei (hier: `peter`) eine Benutzergruppe (hier: `ainf`) und für alle anderen Benutzer des Computers.

Im o. a. Beispiel ist die Datei `hello-1.c` für den Benutzer `peter` les- und schreibbar, für alle Angehörigen der Gruppe `ainf` nur lesbar und für alle anderen Benutzer des Computers ebenfalls nur lesbar. Die Datei `hello-1` (ohne Endung) ist hingegen für den Benutzer `peter` les-, schreib- und ausführbar, für alle Angehörigen der Gruppe `ainf` les- und ausführbar, aber nicht schreibbar. Alle anderen Benutzer des Computer haben für die Datei `hello-1` überhaupt keine Zugriffsrechte.

- Welcher Art der Inhalt der Datei ist, entnimmt Unix dem Inhalt selbst. Man kann sich dies mit Hilfe des Befehls `file` anzeigen lassen:

```
$ file hello-1
hello-1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18,
not stripped
$ file hello-1.c
hello-1.c: ASCII C program text
```

- Eine ausführbare Datei, die Text enthält, ist ein sogenanntes *Shell-Skript*. Der Aufruf eines Shell-Skripts bewirkt i. w. dasselbe, als wenn man den darin enthaltenen Text als Kommandos eingeben würde.
- Ein C-Quelltext enthält i. d. R. *keine* gültigen Unix-Kommandos und kann daher *nicht* „einfach so“ ausgeführt werden.
- Es ist zulässig, aber normalerweise nicht sinnvoll, einer ausführbaren Datei die Endung `.c` zu geben.

## 2.3 Elementare Aus- und Eingabe

Da es möglich ist, mittels der Funktion `printf()` eine String-Konstante wie z. B. `"Hello,_world!\n"` „einfach so“ auszugeben, liegt die Vermutung nahe, Integer-Konstanten auf gleiche Weise ausgeben zu können.

Datei `output-1.c`:

```
#include <stdio.h>

int main (void)
{
    printf (42);
    return 0;
}
```

Beim Compilieren dieses Programms erhalten wir eine Warnung:

```
$ gcc output-1.c -o output-1
output-12.c: In function 'main':
output-12.c:5: warning: passing argument 1 of 'printf'
makes pointer from integer without a cast
/usr/include/stdio.h:339: note: expected
'const char * __restrict__' but argument is of type 'int'
```

Es entsteht trotzdem eine ausführbare Datei `output-1`. Wenn wir diese jedoch ausführen, erhalten wir eine Fehlermeldung:

```
$ ./output-12
Segmentation fault
```

Tatsächlich ist die direkte Übergabe einer Integer-Konstanten an `printf()` ein grober Fehler: `printf()` akzeptiert als ersten Parameter nur Ausdrücke vom Typ String. Der C-Compiler nimmt eine implizite Umwandlung der Integer-Konstanten in einen String vor: Die Zahl wird als eine Speicheradresse interpretiert, an der sich der Text befindet. Dies ist nicht besonders sinnvoll (daher die Warnung), aber in C zulässig.

Wenn nun das Programm ausgeführt wird, versucht es, auf die Speicheradresse Nr. 42 zuzugreifen. Diese befindet sich normalerweise außerhalb des Programms. Das Betriebssystem bemerkt den illegalen Zugriffsversuch und bricht das Programm mit einer Fehlermeldung („Speicherzugriffsfehler“, „Schutzverletzung“ o. ä.) ab.

Auf einer Plattform ohne derartige Schutzmechanismen (z. B. einem Mikrocontroller) wird das fehlerhafte Programm hingegen klaglos ausgeführt. Es werden dann sinnlose Texte, die sich zufällig an Speicheradresse Nr. 42 befinden, auf dem Standardausgabegerät ausgegeben.



Dieses fehlerhafte Programm illustriert, wie leicht es in der Programmiersprache C ist, einen Absturz zu programmieren. Die meisten anderen Programmiersprachen würden das fehlerhafte Programm nicht akzeptieren; anstelle der o. a. Warnung bekäme man eine ähnlichlautende Fehlermeldung.



**Nehmen Sie nicht nur die Fehlermeldungen, sondern auch die Warnungen des Compilers ernst!**

Gerade in graphischen Entwicklungsumgebungen werden Warnungen oft in einem winzigen Fenster angezeigt und gehen zwischen anderen Meldungen unter. Auch sind die Compiler-Optionen, mit denen Sie Warnungen ein- oder ausschalten können, oft in tiefen Menü-Strukturen versteckt, so daß man als Programmierer den Aufwand scheut, diese sinnvoll zu setzen.

Fehlermeldungen *müssen* Sie ernstnehmen, da Sie sonst kein ausführbares Programm erhalten. Warnungen *sollten* Sie ebenfalls ernstnehmen, *obwohl* Sie ein ausführbares Programm erhalten, da dieses mit hoher Wahrscheinlichkeit in einer nicht-offensichtlichen Weise *fehlerhaft* ist. Ein derartiges Programm produktiv einzusetzen, kann je nach Einsatzgebiet Vermögens-, Sach- oder sogar Personenschäden zur Folge haben.

Wie man nun tatsächlich in C Zahlenwerte ausgibt, illustriert das Beispielprogramm [output-2.c](#):

```
#include <stdio.h>

int main (void)
{
    printf ("Die_Antwort_lautet:_%d\n", 42);
    return 0;
}
```

Der erste Parameter von `printf()`, der sog. *Format-String*, enthält das Symbol `%d`. Diese sog. *Formatspezifikation* wird in der Ausgabe durch den Zahlenwert des zweiten Parameters von `printf()` ersetzt. Das `d` steht hierbei für „dezimal“.

Wenn man zwischen das Prozentzeichen und das `d` eine Zahl schreibt (z. B. `%3d`), gibt man damit die Breite eines Feldes an, in die die auszugebende Zahl rechtsbündig geschrieben wird. Wenn man die Feldbreite mit einer Null beginnen läßt (z. B. `%03d`) wird die auszugebende Zahl von links mit Nullen bis zur Feldbreite aufgefüllt.

Eine vollständige Liste der in `printf()` zulässigen Formatspezifikationen finden Sie in der Dokumentation des Compiler-Herstellers zu `printf()`. Von der Unix-Shell aus können Sie diese mit dem Befehl `man 3 printf` abrufen.

Bemerkungen:

- Ein Text darf auch Ziffern enthalten. Anhand der Ausgabe sind `printf ("42\n");` und `printf ("%d\n", 42);` nicht voneinander unterscheidbar.
- Die Position des `\n` ist relevant, z. B. geht `printf ("\n42");` zuerst in eine neue Zeile und gibt danach den Text aus. Auch mehrere `\n` in derselben String-Konstanten sind zulässig.
- C akzeptiert auch sehr seltsame Konstrukte. Das folgende Beispiel (Datei: [hello-2.c](#))

```
#include <stdio.h>

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

wird vom Compiler akzeptiert. (Warum das so ist, wird in Abschnitt [2.7](#) behandelt.)

Bei Verwendung der zusätzlichen Option `-Wall` erhalten wir zumindest eine Warnung über eine „Anweisung ohne Effekt“:

```
$ gcc -Wall hello-2.c -o hello-2
hello-2.c: In function 'main':
hello-2.c:6: warning: statement with no effect
```

Es empfiehlt sich, die Option `-Wall` grundsätzlich zu verwenden und die Warnungen ernstzunehmen.

Wenn mehrere Werte ausgegeben werden sollen, verwendet man in `printf()` mehrere Formatspezifikationen und gibt mehrere Werte als Parameter an (Datei: `output-3.c`):

```
#include <stdio.h>

int main (void)
{
    printf ("Richtige_Antworten_wären_%d_oder_%d_oder_sonstige.\n", 1, 2);
    return 0;
}

$ gcc output-3.c -o output-3
$ ./output-3
Richtige Antworten wären 1 oder 2 oder sonstige.
$
```

Achtung: Zu viele oder zu wenige Werte in der Parameterliste ergeben trotzdem ein gültiges, wenn auch fehlerhaftes C-Programm (Datei: `output-4.c`):

```
#include <stdio.h>

int main (void)
{
    printf ("Richtige_Antworten_wären_%d", 1, "_oder_%d", 2, "_oder_sonstige.\n");
    return 0;
}
```

Wenn man dieses Programm laufen läßt, wird nicht etwa das zweite `%d` durch den Zahlenwert 2 ersetzt. Vielmehr endet das, was ausgegeben wird, mit dem ersten `%d`, für das der Zahlenwert 1 eingesetzt wird, und alles, was nach der 1 kommt, wird schlichtweg ignoriert.

```
$ gcc output-4.c -o output-4
$ ./output-4
Richtige Antworten wären 1$
```

Bei Verwendung der Option `-Wall` erhalten wir auch hier eine Warnung:

```
$ gcc -Wall output-4.c -o output-4
output-4.c: In function 'main':
output-4.c:5: warning: too many arguments for format
```

Das Einlesen von Werten erfolgt in C mit der Funktion `scanf()`.

Das folgende Beispielprogramm (Datei: `input-1.c`) liest einen Wert vom Standardeingabegerät (hier: Tastatur) ein und gibt ihn wieder aus:

```
#include <stdio.h>

int main (void)
{
    int a;
    printf ("Bitte_eine_Zahl_eingeben:_");
    scanf ("%d", &a);
    printf ("Sie_haben_eingegeben:_%d\n", a);
    return 0;
}
```

Damit `scanf()` in die Variable `a` einen Wert schreiben kann, ist es erforderlich, nicht den aktuellen Wert von `a`, sondern die Variable selbst an `scanf()` zu übergeben. Dies geschieht durch Voranstellen eines Und-Symbols `&`. (Genaugenommen handelt es sich um die Übergabe einer Speicheradresse. Dies wird in Abschnitt 2.10 genauer behandelt.)

Wenn wir das `&` vergessen (Beispielprogramm: `input-2.c`), kann das C-Programm weiterhin kompiliert werden. Bei Verwendung der Option `-Wall` erhalten wir eine Warnung. Wenn wir das Programm ausführen und versuchen, einen Wert einzugeben, stürzt das Programm ab. (Hintergrund: Es betrachtet den aktuellen – zufälligen – Wert der Variablen `a` als Adresse einer Speicherzelle, an der der eingelesene Wert gespeichert werden soll. Das Programm greift also schreibend auf eine Speicherzelle außerhalb des ihm zugeteilten Bereichs zu.)

Die Funktion `scanf()` kann, analog zu `printf()`, gleichzeitig mehrere Werte abfragen. Hierzu müssen wir im Format-String mehrere Formatspezifikationen angeben und die Adressen mehrerer Variablen als Parameter übergeben.

Genau wie bei `printf()` werden überzählige Parameter ignoriert, und fehlende Parameter führen zu einem Absturz des Programms.

Zeichen zwischen den Formatspezifikationen fungieren als Trennzeichen. Damit die Zahlen angenommen werden, muß die Eingabe die Trennzeichen enthalten.

Für doppelt genaue Fließkommazahlen (**double**) lautet die Formatspezifikation `%lf`; für einfach genaue Fließkommazahlen (**float**) lautet sie `%f`.

Weitere Informationen zu den Formatspezifikationen von `scanf()` finden Sie in der Dokumentation zu `scanf()`. (In der Unix-Shell können Sie diese mit dem Befehl `man 3 scanf` abrufen.)

## 2.4 Elementares Rechnen

Der *binäre Operator* `+` kann in C (und den meisten Programmiersprachen) dazu verwendet werden, zwei Integer-Ausdrücke, die sogenannten *Operanden*, durch Addition zu einem neuen Integer-Ausdruck zu verknüpfen.

Beispiel: `mathe-1.c`

```
#include <stdio.h>

int main (void)
{
    printf ("%d\n", 23 + 19);
    return 0;
}
```

(Tatsächlich führt bereits die erste Stufe des Compilers eine Optimierung durch, die bewirkt, daß die ausführbare Datei keine Additionsbefehle, sondern direkt das Ergebnis der Addition enthält.)

Die Operatoren für die Grundrechenarten lauten in C:

<code>+</code>	Addition
<code>-</code>	Subtraktion
<code>*</code>	Multiplikation
<code>/</code>	Division: Bei ganzen Zahlen wird grundsätzlich abgerundet.
<code>%</code>	Modulo-Operation: Rest bei Division ( <code>39 % 4</code> ergibt <code>3</code> .)

Die Verwendung von *Variablen* erfordert in C eine vorherige Deklaration.

```
int a;
```

deklariert eine Variable vom Typ Integer,

```
int a, b;
```

deklariert zwei Variable vom Typ Integer, und

```
int a, b = 3;
```

deklariert zwei Variable vom Typ Integer und initialisiert *die zweite* mit dem Wert 3. (Im letzten Beispiel wird insbesondere die erste Variable *a* *nicht* initialisiert.)

Nicht initialisierte Variable erhalten einen *zufälligen* Wert. Wenn beim Compilieren mit *gcc* zusätzlich zu den Warnungen (Option *-Wall*) auch die Optimierung (Option *-O*, *-O2* oder *O3*) aktiviert ist, erkennt *gcc* die Verwendung derartiger zufälliger Werte und gibt eine Warnung aus.

Nicht explizit initialisierte *globale Variable*, also solche, die außerhalb einer Funktion deklariert werden, werden implizit auf Null initialisiert (0 für Zahlen, *NULL* für Zeiger usw.). Es ist trotzdem in Hinblick auf selbstdokumentierenden Quelltext sinnvoll, diese ggf. explizit auf 0 zu initialisieren.

Für Fließkommazahlen verwendet man meistens den Datentyp *double*:

```
double x = 3.141592653589793;
```

Die Bezeichnung *double* steht für „doppelt genau“. Daneben gibt es noch einen Datentyp *float* für „einfach genaue“ Fließkommazahlen sowie einen Datentyp *long double* für noch höhere Genauigkeit. Typischerweise folgen Fließkommazahlen in C dem Standard IEEE 754. In diesem Fall hat *float* eine Genauigkeit von ca. 6 und *double* eine Genauigkeit von ca. 15 Nachkommastellen.

Zuweisungen an Variable erfolgen in C mit Hilfe des binären Operators *=*. Es ist ausdrücklich erlaubt, den „alten“ Wert einer Variablen in Berechnungen zu verwenden, deren Ergebnis man dann derselben Variablen zuweist.

Eine Anweisung wie z. B. *a = 2 \* a* ist insbesondere keine mathematische Gleichung (mit der Lösung 0 für die Unbekannte *a*), sondern die Berechnung des Doppelten des aktuellen Wertes der Variablen *a*, welches dann wiederum in der Variablen *a* gespeichert wird.

## 2.5 Verzweigungen

Das Beispielprogramm *if-0.c*

```
#include <stdio.h>

int main (void)
{
    int a, b;
    printf ("Bitte_a_eingeben:_");
    scanf ("%d", &a);
    printf ("Bitte_b_eingeben:_");
    scanf ("%d", &b);
    printf ("a_geteilt_durch_b_ist:_%d\n", a / b);
    return 0;
}
```

hat den Nachteil, daß bei Eingabe von 0 für die zweite Zahl das Programm abstürzt:

```
$ gcc -Wall if-0.c -o if-0
$ ./if-0
Bitte a eingeben: 13
Bitte b eingeben: 0
Floating point exception
```

Die Fehlermeldung stammt nicht vom Programm selbst, sondern vom Betriebssystem, das auf einen vom Prozessor signalisierten Fehlerzustand reagiert. („Floating point exception“ ist die Bezeichnung dieses Fehlerzustands. In diesem Fall ist die Bezeichnung leicht irreführend, da konkret dieser Fehler durch eine Division ganzer Zahlen, also insbesondere nicht durch eine Fließkommaoperation, ausgelöst wird.)

Für Programme wie dieses ist es notwendig, in Abhängigkeit von den Benutzereingaben unterschiedliche Anweisungen auszuführen. Diese sog. *Verzweigung* geschieht mittels einer *if*-Anweisung.

Beispielprogramm: if-1.c

```
#include <stdio.h>

int main (void)
{
    int a, b;
    printf ("Bitte_a_eingeben:_");
    scanf ("%d", &a);
    printf ("Bitte_b_eingeben:_");
    scanf ("%d", &b);
    if (b != 0)
        printf ("a_geteilt_durch_b_ist:_%d\n", a / b);
    return 0;
}
```

In den Klammern hinter dem **if** steht ein Ausdruck, die sog. *Bedingung*. Die auf das **if** folgende Anweisung wird nur dann ausgeführt, wenn die Bedingung *ungleich Null* ist. (C kennt keinen eigenen „Booleschen“ Datentyp. Stattdessen steht **0** für den Wahrheitswert „falsch“ und alles andere für den Wahrheitswert „wahr“.)

Der binäre Operator **!=** prüft zwei Ausdrücke auf Ungleichheit. Er liefert **0** zurück, wenn beide Operanden gleich sind, und **1**, wenn sie ungleich sind.

Die **if**-Anweisung kennt einen optionalen **else**-Zweig. Dieser wird dann ausgeführt, wenn die Bedingung *nicht* erfüllt ist.

Beispielprogramm: if-2.c

```
#include <stdio.h>

int main (void)
{
    int a, b;
    printf ("Bitte_a_eingeben:_");
    scanf ("%d", &a);
    printf ("Bitte_b_eingeben:_");
    scanf ("%d", &b);
    if (b != 0)
        printf ("a_geteilt_durch_b_ist:_%d\n", a / b);
    else
        printf ("Bitte_nicht_durch_0_teilen!\n");
    return 0;
}
```

Sowohl auf das **if** als auch auf das **else** folgt nur jeweils *eine* Anweisung, die von der Bedingung abhängt.

In dem folgenden Beispielprogramm (Datei: if-3.c)

```
#include <stdio.h>

int main (void)
{
    int a, b;
    printf ("Bitte_a_eingeben:_");
    scanf ("%d", &a);
    printf ("Bitte_b_eingeben:_");
    scanf ("%d", &b);
    if (b != 0)
        printf ("a_geteilt_durch_b_ist:_%d\n", a / b);
    else
        printf ("Bitte_nicht_durch_0_teilen!\n");
        printf ("Das_tut_man_nicht.\n");
    return 0;
}
```

wird die Zeile `printf ("Das_tut_man_nicht.\n");` auch dann ausgeführt, wenn die Variable `b` ungleich 0 ist.

In C ist die Einrückung der Zeilen im Programm Quelltext „nur“ eine optische Hilfe für Programmierer. Welche Anweisung von welcher Bedingung abhängt, entscheidet der Compiler allein anhand der Regeln der Programmiersprache, und diese besagen eindeutig: „Sowohl auf das **if** als auch auf das **else** folgt nur jeweils *eine* Anweisung, die von der Bedingung abhängt.“

Wenn wir möchten, daß mehrere Anweisungen von der Bedingung abhängen, müssen wir diese mittels geschweifeter Klammern zu einem sog. *Anweisungsblock* zusammenfassen.

Beispielprogramm: `if-4.c`

```
#include <stdio.h>

int main (void)
{
    int a, b;
    printf ("Bitte_a_eingeben:_");
    scanf ("%d", &a);
    printf ("Bitte_b_eingeben:_");
    scanf ("%d", &b);
    if (b != 0)
        printf ("a_geteilt_durch_b_ist:_%d\n", a / b);
    else
    {
        printf ("Bitte_nicht_durch_0_teilen!\n");
        printf ("Das_tut_man_nicht.\n");
    }
    return 0;
}
```

Aus Sicht des Computers ist die Einrückung – und überhaupt die Anordnung von Leerzeichen und Zeilenschaltungen – belanglos. Die folgende Schreibweise (Datei: `if-5.c`) ist für ihn vollkommen gleichwertig zu `if-4.c`:

```
#include<stdio.h>
int main(void){int a,b;printf("Bitte_a_eingeben:_");scanf("%d",&a);
printf("Bitte_b_eingeben:_");scanf("%d",&b);if(b!=0)printf(
"a_geteilt_durch_b_ist:_%d\n",a/b);else{printf("Bitte_nicht_durch_0_teilen!\n");
printf("Das_tut_man_nicht.\n");}return 0;}
```

Aus Sicht eines Menschen hingegen kann eine *korrekte* Einrückung des Quelltextes *sehr* hilfreich dabei sein, in einem Programm die Übersicht zu behalten.

Daher hier der dringende Rat:



**Achten Sie in Ihren Programmen auf korrekte und übersichtliche Einrückung!**

Um zwei Ausdrücke auf Gleichheit zu prüfen, verwendet man in C den binären Operator `==`.

Die Anweisungen

```
if (b != 0)
{
    printf ("Die_erste_Zahl_geteilt_durch_die_zweite_ergibt:_");
    printf ("%d,_Rest_%d\n", a / b, a % b);
}
else
    printf ("Bitte_nicht_durch_0_teilen!\n");
```

sind also äquivalent zu:

```
if (b == 0)
    printf ("Bitte_nicht_durch_0_teilen!\n");
else
```

```

{
    printf ("Die_erste_Zahl_geteilt_durch_die_zweite_ergibt:_");
    printf ("%d,_Rest_%d\n", a / b, a % b);
}

```

Achtung: Die Anweisungen

```

if (b = 0)
    printf ("Bitte_nicht_durch_0_teilen!\n");
else
{
    printf ("Die_erste_Zahl_geteilt_durch_die_zweite_ergibt:_");
    printf ("%d,_Rest_%d\n", a / b, a % b);
}

```

(mit `=` anstelle von `==`) sind ebenfalls gültiges C, haben jedoch eine andere Bedeutung!

Der Hintergrund ist der folgende: Alle binären Operatoren, sei es `+` oder `=` oder `==`, sind in C vom Prinzip her gleichwertig. Alle nehmen zwei numerische Operanden entgegen und liefern einen numerischen Wert zurück. Wenn wir nun beispielsweise annehmen, daß die Variable `a` den Wert 3 hat, dann gilt:

```

a + 7   ergibt 10.
a = 7   ergibt 7 (und weist a den Wert 7 zu).
a == 7  ergibt 0.

```

Das o. a. Programmfragment bedeutet demnach: Weise der Variablen `b` den Wert `0` zu, und führe anschließend *immer* eine Division durch `b` aus. (Die `if`-Bedingung bekommt den Wert `0`, ist also niemals erfüllt.)

Daß es sich bei Wahrheitswerten in C tatsächlich um Integer-Werte handelt, wird auch deutlich, wenn man sich diese mittels `printf()` ausgeben läßt.

Wenn man beispielsweise in dem folgenden Programm `if-6.c` den Wert `7` für die Variable `b` eingibt,

```

#include <stdio.h>

int main (void)
{
    int b;
    printf ("Bitte_b_eingeben:_");
    scanf ("%d", &b);
    printf ("Der_Ausdruck_b!=_0_hat_den_Wert_%d\n", b != 0);
    printf ("Der_Ausdruck_b==_0_hat_den_Wert_%d\n", b == 0);
    printf ("Der_Ausdruck_b=_23_hat_den_Wert_%d\n", b = 23);
    return 0;
}

```

lautet die Ausgabe:

```

$ ./if-6
Bitte b eingeben: 7
Der Ausdruck b != 0 hat den Wert 1
Der Ausdruck b == 0 hat den Wert 0
Der Ausdruck b = 23 hat den Wert 23

```

In der ersten und zweiten Zeile wird geprüft, ob `b` den Wert `0` hat, und `1` für „ja“ bzw. `0` für „nein“ ausgegeben. In der dritten Zeile wird `b` der Wert `23` zugewiesen und anschließend der neue Wert von `b` ausgegeben.

## 2.6 Schleifen

Mit Hilfe der `while`-Anweisung ist es möglich, Anweisungen in Abhängigkeit von einer Bedingung mehrfach auszuführen.

Das folgende Beispielprogramm `loop-1.c` schreibt die Zahlen von 1 bis einschließlich 10 auf den Bildschirm:

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

```
int main (void)
{
    int i = 1;
    while (i <= 10)
    {
        printf ("%d\n", i);
        i = i + 1;
    }
    return 0;
}
```

Die Auswertung der Bedingung erfolgt analog zur **if**-Anweisung. Ebenso folgt auf **while** nur eine einzige Anweisung, die wiederholt ausgeführt wird; mehrere Anweisungen müssen mit geschweiften Klammern zu einem Anweisungsblock zusammengefaßt werden.

Der binäre Operator **<=** liefert 1 zurück, wenn der linke Operand kleiner oder gleich dem rechten ist, ansonsten 0. Entsprechend sind die Operatoren **>=**, **<** und **>** definiert.

Ein wichtiger Spezialfall einer **while**-Schleife ist die folgende Situation:

- Vor dem Betreten der Schleife findet eine Initialisierung statt, z. B. **i = 1**.
- Am Ende jedes Schleifendurchlaufs wird eine „Schritt-Anweisung“ durchgeführt, z. B. **i = i + 1**.

Für dieses spezielle **while** kennt C die Abkürzung **for**:

<pre>int i = 1; while (i &lt;= 10) {     printf ("%d\n", i);     i = i + 1; }</pre>	ist genau dasselbe wie	<pre>int i; for (i = 1; i &lt;= 10; i = i + 1)     printf ("%d\n", i);  oder  for (int i = 1; i &lt;= 10; i = i + 1)     printf ("%d\n", i);</pre>
(Datei: <a href="#">loop-2.c</a> )		

Achtung: Zwischen den Klammern nach **for** stehen zwei Semikolons, keine Kommata.



**Die Schreibweise mit der Deklaration `int i = 1` innerhalb der `for`-Schleife ist erst ab dem C-Standard C99 zulässig. Beim Compilieren mit älteren Versionen des `gcc` muß daher zusätzlich die Option `-std=c99` angegeben werden.**

Als eine weitere Schleife kennt C die **do-while**-Schleife:

```
i = 1;
do
{
    printf ("%d\n", i);
    i = i + 1;
}
while (i <= 10)
```

Der Unterschied zur „normalen“ **while**-Schleife besteht darin, daß eine **do-while**-Schleife mindestens einmal ausgeführt wird, weil die Bedingung nicht bereits am Anfang, sondern erst am Ende des Schleifendurchlaufs geprüft wird.



Zwischen einer „normalen“ **while**-Schleife und einer **for**-Schleife besteht hingegen *kein* Unterschied. Insbesondere ist eine Schreibweise wie

```
for (i = 1; 10; i + 1)
    printf ("%d\n", i);
```

zwar zulässiges C, aber nicht sinnvoll (Datei: [loop-3.c](#)). Dies kann man sofort erkennen, indem man die **for**-Schleife in eine **while**-Schleife übersetzt:

```
i = 1;
while (10)
{
    printf ("%d\n", i);
    i + 1;
}
```

Dieses Programmfragment setzt einmalig *i* auf den Wert 1 und springt danach in eine Endlosschleife zur Ausgabe von *i*. (Die **while**-Bedingung 10 ist ungleich Null, hat also stets den Wahrheitswert „wahr“.) Am Ende jedes Schleifendurchlaufs wird *i* + 1 berechnet; der berechnete Wert wird jedoch nirgendwo verwendet, sondern schlichtweg verworfen. Insbesondere ändert *i* seinen Wert nicht.

## 2.7 Seiteneffekte

Das Verwerfen berechneter Werte verdient eine nähere Betrachtung – insbesondere in der Programmiersprache C. Wie das Beispielprogramm [statements-1.c](#) illustriert,

```
#include <stdio.h>

int main (void)
{
    2 + 2;
    return 0;
}
```

ist es anscheinend zulässig, Werte als Anweisung zu verwenden.

Grundsätzlich gilt in C: Man kann jeden gültigen Ausdruck als Anweisung verwenden. Der Wert des Ausdrucks wird dabei ignoriert.

Die Bedeutung der (gültigen!) C-Anweisung `2 + 2`; lautet somit: „Berechne den Wert `2 + 2` und vergiß ihn wieder.“

Tatsächlich gilt dasselbe auch für `printf()`: Die Funktion `printf()` liefert eine ganze Zahl zurück. Der `printf()`-Aufruf ist somit ein Ausdruck, dessen Wert ignoriert wird.

„Nebenbei“ hat `printf()` aber noch eine weitere Bedeutung, nämlich die Ausgabe des Textes auf dem Standardausgabegerät (Bildschirm). Diese weitere Bedeutung heißt *Seiteneffekt* des Ausdrucks.

Das Beispielprogramm [statements-2.c](#) gibt den vom ersten `printf()` zurückgegebenen Wert mit Hilfe eines zweiten `printf()` aus:

```
#include <stdio.h>

int main (void)
{
    int x;
    x = printf ("%d\n", 2 + 2);
    printf ("%d\n", x);
    return 0;
}
```

Die Ausgabe lautet:

```
4
2
```

Bei dem von `printf()` zurückgegebenen Wert handelt es sich um die Anzahl der geschriebenen Zeichen. In diesem Fall ist es zwei Zeichen, nämlich die Ziffer `4` sowie das Zeilenendesymbol, im Programm als `\n` notiert.

Auch Operatoren können in C Seiteneffekte haben.

- Der binäre Operator `=` (Zuweisung) hat als Seiteneffekt die Zuweisung des zweiten Operanden an den ersten Operanden und als Rückgabewert den zugewiesenen Wert.
- Ähnlich funktionieren die binären Operatoren `+=` `-=` `*=` `/*=`. Sie wenden die vor dem `=` stehende Rechenoperation auf die beiden Operanden an, weisen als Seiteneffekt das Ergebnis dem ersten Operanden zu und geben das Rechenergebnis als Wert zurück.
- Die binären Rechenoperatoren `+` `-` `*` `/` `%` und Vergleichsoperatoren `==` `!=` `<` `>` `<=` `>=` haben *keinen* Seiteneffekt.
- Der unäre Rechenoperator `-` (arithmetische Negation, Vorzeichen) hat ebenfalls *keinen* Seiteneffekt.
- Ein weiterer unärer Operator *ohne* Seiteneffekt ist die logische Negation, in C ausgedrückt durch ein Ausrufezeichen: `!`  
`!a` ist 1, wenn `a` den Wert 0 hat; ansonsten ist es 0.  
`!(a < b)` ist demzufolge dasselbe wie `a >= b`.
- Der Funktionsaufruf `()` (Klammerpaar) ist in C ebenfalls ein unärer Operator. Er liefert einen Wert zurück (Rückgabewert der Funktion) und hat einen Seiteneffekt (Aufruf der Funktion).
- Die unären Operatoren `++` und `--` haben den Seiteneffekt, daß sie die Variable, vor oder hinter der sie stehen, um 1 erhöhen (`++`) bzw. vermindern (`--`). Wenn der Operator *vor* der Variablen steht (`++a`), ist der Rückgabewert der um 1 erhöhte/verminderte Wert der Variablen. Wenn er hingegen *hinter* der Variablen steht (`a++`), ist der Rückgabewert der ursprüngliche Wert der Variablen; das Erhöhen/Vermindern findet in diesem Fall erst danach statt.  
Da die Reihenfolge, in der ein Ausdruck ausgewertet wird, nicht immer festliegt, sollte man darauf achten, daß die Seiteneffekte eines Ausdruck dessen Wert nicht beeinflussen. (`gcc` warnt in derartigen Fällen.)
- Ein weiterer binärer Operator *ohne* Seiteneffekt ist das Komma. Der Ausdruck `a, b` bedeutet: „Berechne `a`, vergiß es wieder, und gib stattdessen `b` zurück.“ Dies ist nur dann sinnvoll, wenn der Ausdruck `a` einen Seiteneffekt hat.

Die folgenden vier Programmfragmente sind verschiedene Schreibweisen für genau denselben Code.

```
int i;

i = 0;
while (i < 10)
{
    printf ("%d\n", i);
    i += 1;
}

for (i = 0; i < 10; i++)
    printf ("%d\n", i);

i = 0;
while (i < 10)
    printf ("%d\n", i++);

for (i = 0; i < 10; printf ("%d\n", i++));
```

Sie bewirken nicht nur dasselbe (Ausgabe der Zahlen von 0 bis 9), sondern stehen tatsächlich für *genau dasselbe Programm*. Sie laufen genau gleich schnell und unterscheiden sich nur hinsichtlich ihrer Lesbarkeit, wobei es vom persönlichen Geschmack abhängt, welche Variante man jeweils als lesbarer empfindet.



**Schreiben Sie Ihre Programme stets so lesbar wie möglich.  
Platzsparende Schreibweise macht ein Programm nicht schneller.**

## 2.8 Strukturierte Programmierung

Bei den bisher vorgestellten Verzweigungen und Schleifen ist die Reihenfolge, in der die Befehle abgearbeitet werden, klar erkennbar. Darüberhinaus kennt C auch Anweisungen, die einen Sprung des Programms bewirken, der diese Struktur durchbricht:

- Mit der **break**-Anweisung kann das Programm die nächstäußere **while**- oder **for**-Schleife unmittelbar verlassen.

Das folgende Beispielprogramm zählt von 0 bis 9, indem es eine Endlosschleife beim Erreichen von 10 mittels **break** unterbricht. Der Schleifenzähler **i** wird innerhalb des **printf()** „nebenbei“ inkrementiert.

```
int i = 0;
while (1)
{
    if (i >= 10)
        break;
    printf ("%d\n", i++);
}
```

Eine übersichtlichere Schreibweise derselben Schleife lautet:

```
for (int i = 0; i < 10; i++)
    printf ("%d\n", i++);
```

(Der erzeugte Code ist in beiden Fällen genau derselbe.)

- Mit der **continue**-Anweisung springt ein Programm unmittelbar in den nächsten Durchlauf der nächst-äußeren Schleife.
- Mit der **return**-Anweisung kann man eine Funktion (siehe Abschnitt 2.9) ohne Umweg direkt verlassen.
- Mit der **goto**-Anweisung springt ein Programm direkt an einen *Label*. Dieser besteht aus einem Namen, gefolgt von einem Doppelpunkt.

```
int i = 0;
loop:
    if (i >= 10)
        goto endloop;
    printf ("%d\n", i++);
    goto loop;
endloop:
```

Ein Programmquelltext sollte immer so gestaltet werden, daß er den Ablauf des Programms unmittelbar ersichtlich macht. Ein vorzeitiges **return** stellt einen „Hinterausgang“ einer Funktion dar und sollte mit Bedacht eingesetzt werden.

Ähnliches gilt in noch stärkerem Maße für **break** und **continue** als „Hinterausgänge“ von Schleifen. Diese sind sicherlich bequeme Möglichkeiten, zusätzliche **ifs** und zusätzliche Wahrheitswert-Variable zu vermeiden, verschleiern aber langfristig den Ablauf der Befehle. Statt eine Schleife mit **break** zu verlassen oder Teile des Schleifeninneren mit **continue** zu überspringen, ist es besser, die Schleifenbedingung und **if**-Anweisungen innerhalb der Schleife so zu formulieren, daß Sie kein **break** oder **continue** mehr benötigen. Dadurch versteht man auch selbst besser, was das Programm eigentlich tut. Das Programm wird übersichtlicher und oft sogar kürzer.

In besonderem Maße gilt dies für die **goto**-Anweisung. Hier ist nicht erkennbar, ob der Sprung nach oben geht (Schleife) oder nach unten (Verzweigung). Verschachtelungen von Blöcken und **goto**-Sprüngen bereiten dem Compiler zusätzliche Arbeit und stehen somit der Optimierung entgegen. (Es stimmt insbesondere nicht, daß Konstruktionen mit **goto** schneller abgearbeitet würden als Konstruktionen mit **if** und **while**.) Es ist daher besser, **goto** nicht zu verwenden und stattdessen den Programmablauf mit Hilfe von Verzweigungen und Schleifen zu strukturieren. (Siehe auch: <http://xkcd.org/292/>)

Zusammengefaßt:



Verwenden Sie vorzeitiges **return** mit Bedacht.

Vermeiden Sie die Verwendung von **break** und **continue**.

Verwenden Sie kein **goto**.

## 2.9 Funktionen

Eine Funktionsdeklaration hat in C die Gestalt:

```
Typ Name ( Parameterliste )
{
    Anweisungen
}
```

Beispielprogramm: [functions-1.c](#)

```
#include <stdio.h>

void foo (int a, int b)
{
    printf ("foo():_a=_%d,_b=_%d\n", a, b);
}

int main (void)
{
    foo (3, 7);
    return 0;
}
```

(Das Wort „foo“ ist eine sog. *metasyntaktische Variable* – ein Wort, das absichtlich nichts bedeutet und für einen beliebig austauschbaren Namen steht.)

Mit dem Funktionsaufruf `foo (3, 7)` stellt das Hauptprogramm der Funktion `foo()` die Parameterwerte 3 für `a` und 7 für `b` zur Verfügung.

Der Rückgabewert der Funktion `foo()` ist vom Typ `void`. Im Gegensatz zu Datentypen wie z. B. `int`, das für ganze Zahlen steht, steht `void` für „nichts“.

Von Ausdrücken zurückgegebene `void`-Werte *müssen* ignoriert werden. (Von Ausdrücken zurückgegebene Werte anderer Typen *dürfen* ignoriert werden.)

Das Hauptprogramm ist in C eine ganz normale Funktion. Dadurch, daß sie den Namen `main` hat, weiß das Betriebssystem, daß es sie bei Programmbeginn aufrufen soll. `main()` kann dann seinerseits weitere Funktionen aufrufen.

Über seinen Rückgabewert (vom Typ `int`) teilt `main()` dem Betriebssystem mit, ob das Programm erfolgreich beendet werden konnte. Der Rückgabewert 0 steht für „Erfolg“; andere Werte stehen für verschiedenartige Fehler.

Je nachdem, wo und wie Variable deklariert werden, sind sie von verschiedenen Stellen im Programm aus zugänglich und/oder verhalten sich unterschiedlich.

Beispielprogramm: [functions-2.c](#)

```
1  #include <stdio.h>
2
3  int a, b = 3;
4
5  void foo (void)
6  {
7      b++;
8      static int a = 5;
9      int b = 7;
10     printf ("foo():_a=_%d,_b=_%d\n", a, b);
11     a++;
12     b++;
13 }
14
15 int main (void)
```

```

16 {
17     printf ("main():_a=_%d,_b=_%d\n", a, b);
18     foo ();
19     printf ("main():_a=_%d,_b=_%d\n", a, b);
20     a = b = 12;
21     printf ("main():_a=_%d,_b=_%d\n", a, b);
22     foo ();
23     printf ("main():_a=_%d,_b=_%d\n", a, b);
24     return 0;
25 }

```

Die Ausgabe dieses Programms lautet:

```

main(): a = 0, b = 3
foo(): a = 5, b = 7
main(): a = 0, b = 4
main(): a = 12, b = 12
foo(): a = 6, b = 7
main(): a = 12, b = 13

```

Erklärung:

- Der erste Aufruf der Funktion `printf()` in Zeile 17 des Programms gibt die Werte der in Zeile 3 deklarierten Variablen aus. Diese lauten 0 für `a` und 3 für `b`.  
Weil es sich um sog. *globale Variable* handelt (Die Deklaration steht außerhalb jeder Funktion.), werden diese Variablen *bei Programmbeginn* initialisiert. Für `b` steht der Wert 3 für die Initialisierung innerhalb der Deklaration; für `a` gilt der implizite Wert 0.
- Der zweite Aufruf von `printf()` erfolgt indirekt über die Funktion `foo()`, die ihrerseits vom Hauptprogramm aus aufgerufen wurde (Zeile 18).  
Oberhalb des `printf()` (Zeile 10) befinden sich neue Deklarationen für Variable, die ebenfalls `a` (Zeile 8) und `b` heißen (Zeile 9). Diese sog. *lokalen Variablen* werden auf neue Werte initialisiert, die korrekt ausgegeben werden.  
Ab den Zeilen 8 und 9 bis zum Ende der Funktion `foo()` sind die in Zeile 3 deklarierten globalen Variablen `a` und `b` nicht mehr zugreifbar.
- Der dritte Aufruf von `printf()` erfolgt wieder direkt durch das Hauptprogramm (Zeile 19).  
`a` hat immer noch den Wert 0, weil durch das `a++` in Zeile 11 eine andere Variable inkrementiert wurde, die ebenfalls `a` heißt, nämlich die lokale Variable, die in Zeile 8 deklariert wurde.  
Dasselbe gilt für `b` hinsichtlich der Zeile 12. In Zeile 7 jedoch greift die Funktion `foo()` auf die in Zeile 3 deklarierte globale Variable `b` zu, die dadurch den Wert 4 (statt vorher: 3) erhält.
- In Zeile 20 weist das Hauptprogramm beiden in Zeile 3 deklarierten Variablen den Wert 12 zu.  
Genauer: Es weist der Variablen `a` den Wert `b = 12` zu. Bei `b = 12` handelt es sich um einen Ausdruck mit Seiteneffekt, nämlich die Zuweisung des Wertes 12 an die Variable `b`. Der Wert des Zuweisungsausdrucks ist ebenfalls 12.
- Der vierte Aufruf von `printf()` erfolgt wieder direkt durch das Hauptprogramm (Zeile 21) und gibt erwartungsgemäß zweimal den Wert 12 aus.
- Der fünfte Aufruf von `printf()` erfolgt wieder indirekt über die Funktion `foo()`, die ihrerseits vom Hauptprogramm aus aufgerufen wurde (Zeile 22).  
Die Funktion `foo()` gibt wiederum die Werte der in den Zeilen 8 und 9 deklarierten Variablen aus.  
Bei `b` (Zeile 9) handelt es sich um eine *automatische Variable*. Diese ist nur innerhalb des umgebenden Blockes – hier der Funktion `foo()` – bekannt. Sie wird beim Aufruf der Funktion initialisiert und hat daher in Zeile 10 stets den Wert 7, den sie in Zeile 9 bekommen hat.  
Die Variable `a` (Zeile 8) ist hingegen als *statisch* (engl. **static**) deklariert. Sie behält ihren Wert zwischen zwei Aufrufen von `foo()`, wird nur zu Programmbeginn initialisiert und ist von außerhalb der Funktion nicht veränderbar.  
Ausnahme: Wenn einer anderen Funktion die Adresse der **static**-Variablen bekannt ist, kann diese die Variable über einen Zeiger verändern – Siehe Abschnitt 2.10.

Da der Anfangswert 5 der Variablen `a` bereits einmal erhöht wurde (Zeile 11), wird der Wert 6 ausgegeben. (Die Zuweisung des Wertes 12 im Hauptprogramm bezog sich auf ein anderes `a`, nämlich das in Zeile 3 deklarierte.)

- Der letzte Aufruf von `printf()` erfolgt wieder direkt durch das Hauptprogramm (Zeile 23).  
`a` hat immer noch den Wert 12, weil durch das `a++` in Zeile 11 eine andere Variable inkrementiert wurde, die ebenfalls `a` heißt, nämlich die, die in Zeile 8 deklariert wurde.  
Dasselbe gilt für `b` hinsichtlich der Zeile 12. In Zeile 7 jedoch greift die Funktion `foo()` auf die in Zeile 3 deklarierte Variable `b` zu, die dadurch den Wert 13 (statt vorher: 12) erhält.

## 2.10 Zeiger

In C können an Funktionen grundsätzlich nur Werte übergeben werden. Vom Funktionsrückgabewert abgesehen, hat eine C-Funktion keine Möglichkeit, dem Aufrufer Werte zurückzugeben.

Es ist dennoch möglich, eine C-Funktion aufzurufen, um eine Variable (oder mehrere) auf einen Wert zu setzen. Hierfür übergibt man der Funktion die *Speicheradresse* der Variablen als Wert. Der Wert ist ein *Zeiger* auf die Variable.

Wenn einem Zeiger der unäre Operator `*` vorangestellt wird, ist der resultierende Ausdruck diejenige Variable, auf die der Zeiger zeigt. In Deklarationen wird dasselbe Symbol dem Namen vorangestellt, um anstelle einer Variablen des genannten Typs eine Variable vom Typ „Zeiger auf Variable des genannten Typs“ zu deklarieren. (Das `*`-Symbol wirkt jeweils nur auf den unmittelbar folgenden Bezeichner.)

Umgekehrt wird der unäre Operator `&` einer Variablen vorangestellt, um einen Ausdruck vom Typ „Zeiger auf Variable dieses Typs“ mit dem Wert „Speicheradresse dieser Variablen“ zu erhalten.

Beispielprogramm: [pointers-1.c](#)

```
#include <stdio.h>

void calc_answer (int *a)
{
    *a = 42;
}

int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

Die Funktion `calc_answer()` läßt sich vom Hauptprogramm einen Zeiger `a` auf die lokale Variable `answer` des Hauptprogramms übergeben. (Aus Sicht des Hauptprogramms ist dieser Zeiger die Adresse `&answer` der lokalen Variablen `answer`.) Sie schreibt einen Wert in die Variable `*a`, auf die der Zeiger `a` zeigt. Das Hauptprogramm kann diesen Wert anschließend seiner Variablen `answer` entnehmen und mit `printf()` ausgeben.

Vergißt man beim Aufruf den Adreßoperator `&`, übergibt man den aktuellen Wert der Variablen (hier: eine Zahl) anstelle eines Zeigers (und erhält eine Warnung durch den Compiler). Dieser Wert wird als eine Speicheradresse interpretiert. Diese befindet sich in der Regel außerhalb des Bereichs, den das Betriebssystem dem Programm zugewiesen hat. Ein Versuch der Funktion, auf diese Speicheradresse zuzugreifen, führt dann zum Absturz des Programms (Speicherschutzverletzung).

## 2.11 Arrays und Strings

### 2.11.1 Arrays

In C ist es möglich, mit einem Zeiger Arithmetik zu betreiben, so daß er nicht mehr auf die ursprüngliche Variable zeigt, sondern auf ihren Nachbarn im Speicher.

Solche Nachbarn gibt es dann, wenn mehrere Variable gleichen Typs gemeinsam angelegt werden. Eine derartige Ansammlung von Variablen gleichen Typs heißt **Array** (Feldvariable, Vektor).

Beispielprogramm: **arrays-1.c**

```
#include <stdio.h>

int main (void)
{
    int prime[5] = { 2, 3, 5, 7, 11 };
    int *p = prime;
    int i;
    for (i = 0; i < 5; i++)
        printf ("%d\n", *(p + i));
    return 0;
}
```

Die initialisierte Variable **prime** ist ein Array von fünf ganzen Zahlen. Der Bezeichner **prime** des Arrays wird als Zeiger auf eine **int**-Variable verwendet. In diesem Sinne sind Arrays und Zeiger in C dasselbe.

**p + i** ist ein Zeiger auf den **i**-ten Nachbarn von **\*p**. Durch Dereferenzieren **\*(p + i)** erhalten wir den **i**-ten Nachbarn von **\*p** selbst.

Da diese Kombination – Zeigerarithmetik mit anschließendem Dereferenzieren – sehr häufig auftritt, stellt C für die Konstruktion **\*(p + i)** die Abkürzung **p[i]** zur Verfügung.

Die von anderen Sprachen her bekannte Schreibweise **p[i]** für das **i**-te Element eines Arrays **p** ist also in C lediglich eine Abkürzung für **\*(p + i)**, wobei man **p** gleichermaßen als Array wie als Zeiger auffassen kann.

Wenn wir uns dieser Schreibweise bedienen und anstelle des Zeigers **p**, der durchgehend den Wert **prime** hat, direkt **prime** verwenden, erhalten wir das Beispielprogramm **arrays-2.c**:

```
#include <stdio.h>

int main (void)
{
    int prime[5] = { 2, 3, 5, 7, 11 };
    int *p = prime;
    int i;
    for (i = 0; i < 5; i++)
        printf ("%d\n", p[i]);
    return 0;
}
```

Achtung: C prüft *nicht*, ob der Array-Index innerhalb des zulässigen Bereichs liegt, ob also der durch Addition des Index auf die Array-Adresse erhaltene Zeiger noch auf eine Adresse innerhalb des Arrays zeigt.

Übergelaufene Indizes führen nicht immer sofort zum Absturz des Programms, sondern können z. B. andere Variablen des Programms überschreiben. Da derartige Fehler äußerst schwer zu entdecken sind, lohnt es sich, Array-Indices vor ihrer Verwendung mit Hilfe von **if**-Anweisungen „von Hand“ zu prüfen.

## 2.11.2 Strings

Ein wichtiger Spezialfall ist ein Array, dessen Komponenten den Datentyp **char** haben. In C ist **char** wie **int** eine ganze Zahl; der einzige Unterschied besteht darin, daß der Wertebereich von **char** daran angepaßt ist, ein Zeichen (Buchstabe, Ziffer, Satz- oder Sonderzeichen, engl. character) aufzunehmen. Ein typischer Wertebereich für den Datentyp **char** ist von –128 bis 127.

Ein Initialisierer für ein Array von **char**-Variablen kann direkt als Folge von Zeichen (Zeichenkette, engl. *String*) mit doppelten Anführungszeichen geschrieben werden. Jedes Zeichen initialisiert eine ganzzahlige Variable mit seinem ASCII-Wert. An das Ende eines in dieser Weise notierten Array-Initialisierers fügt der Compiler implizit einen Ganzzahl-Initialisierer für den Zahlenwert 0 an. Der Array-Initialisierer **"Hello"** ist also gleichbedeutend mit **{ 72, 101, 108, 108, 111, 0 }**. (Die 72 steht für ein großes H, die 111 für ein kleines o. Man beachte die abschließende 0 am Ende!)

Ein String in C ist also ein Array von **chars**, also ein Zeiger auf **chars**, also ein Zeiger auf ganze Zahlen, deren Wertebereich daran angepaßt ist, Zeichen aufzunehmen.

Wenn bei der Deklaration eines Arrays die Länge aus dem Initialisierer hervorgeht, braucht diese nicht ausdrücklich angegeben zu werden. In diesem Fall folgt auf den Bezeichner nur das Paar eckiger Klammern und der Initialisierer.

Das Beispielprogramm [strings-1.c](#) zeigt, wie das Array durchlaufen werden kann, bis die Zahl 0 gefunden wird:

```
#include <stdio.h>

int main (void)
{
    char hello_world[] = "Hello,_world!\n";
    int i = 0;
    while (hello_world[i] != 0)
        printf ("%d", hello_world[i++]);
    return 0;
}
```

Durch die Formatangabe **%d** wird jedes Zeichen – korrektermaßen – als Dezimalzahl ausgegeben. Wenn wir stattdessen die Formatangabe **%c** verwenden (für *character*), wird für jedes Zeichen – ebenso korrektermaßen – sein Zeichenwert (Buchstabe, Ziffer, ...) ausgegeben (Datei: [strings-2.c](#)):

```
#include <stdio.h>

int main (void)
{
    char hello_world[] = "Hello,_world!\n";
    int i = 0;
    while (hello_world[i] != 0)
        printf ("%c", hello_world[i++]);
    return 0;
}
```

Durch Verwendung von Pointer-Arithmetik und Weglassen der überflüssigen Abfrage **!= 0** erhalten wir das äquivalente Beispielprogramm [strings-3.c](#):

```
#include <stdio.h>

int main (void)
{
    char hello_world[] = "Hello,_world!\n";
    char *p = hello_world;
    while (*p)
        printf ("%c", *p++);
    return 0;
}
```

Dieses ist die in C übliche Art, eine Schleife zu schreiben, die nacheinander alle Zeichen in einem String bearbeitet.

Eine weitere Formatangabe **%s** dient in **printf()** dazu, direkt einen kompletten String bis ausschließlich der abschließenden 0 auszugeben.

Beispielprogramm: [strings-4.c](#)

```
#include <stdio.h>

int main (void)
{
    char *p = "Hello,_world!";
    printf ("%s\n", p);
    return 0;
}
```



Anstatt als Array, das dann einem Zeiger zugewiesen wird, deklarieren wir die Variable `hello_world` direkt als Zeiger. Dies ist die in C übliche Art, mit String-Konstanten umzugehen.

Allein die Formatspezifikation entscheidet darüber, wie die Parameter von `printf()` bei der Ausgabe dargestellt werden:

- `%d` Der Parameter wird als Zahlenwert interpretiert und dezimal ausgegeben.
- `%x` Der Parameter wird als Zahlenwert interpretiert und hexadezimal ausgegeben.
- `%c` Der Parameter wird als Zahlenwert interpretiert und als Zeichen ausgegeben.
- `%s` Der Parameter wird als Zeiger interpretiert und als Zeichenfolge ausgegeben.

## 2.12 String-Operationen

Mit `#include <string.h>` steht uns eine Sammlung von Funktionen zur Bearbeitung von Strings (= Array von `char`-Variablen  $\approx$  Zeiger auf `char`-Variable) zur Verfügung:

### ; ) +-Operationen

Durch Addieren einer ganzen Zahl auf die Startadresse des Strings entsteht ein Zeiger auf einen neuen String, der erst ein paar Zeichen später beginnt. Auf diese Weise kann man in C ganz ohne Benutzung einer Bibliothek den Anfang eines Strings abschneiden.

```
char hello[] = "Hello,_world!\n";
printf ("%s\n", hello + 7);
```

**Achtung:** Es findet keinerlei Prüfung statt, ob der Zeiger nach der Addition noch auf einen Bereich innerhalb des Strings zeigt. Wenn man auf diese Weise über den String hinausliest, führt dies zu unsinnigen Ergebnissen bis hin zu einem Absturz (Speicherzugriffsfehler).

Beispielprogramm: [strings-14.c](#)

### ; ) Null-Zeichen in den String schreiben

Durch das Schreiben eines Null-Symbols (Zahlenwert 0) in den String kann man diesen ganz ohne Benutzung einer Bibliothek an dieser Stelle abschneiden.

```
char hello[] = "Hello,_world!\n";
hello[5] = 0;
printf ("%s\n", hello);
```

**Achtung:** Es findet keinerlei Prüfung statt, ob der Schreibvorgang noch innerhalb des Strings stattfindet. Wenn man auf diese Weise über den String hinaus schreibt, werden andere Variable überschrieben, was in der Regel zu einem Absturz führt (Speicherzugriffsfehler).

Beispielprogramm: [strings-14.c](#)

- `strlen()` – Ermitteln der Länge eines Strings

Das abschließende Null-Symbol wird für die Länge *nicht* mitgezählt, es verbraucht aber natürlich dennoch Speicherplatz.

```
char hello[] = "Hello,_world!\n";
printf ("%s\n", strlen (hello));
```

Beispielprogramm: [strings-14.c](#)

- `strcmp()` – Strings vergleichen

Wenn der erste String-Parameter alphabetisch vor dem zweiten liegt, gibt `strcmp()` den Wert `-1` zurück, wenn es umgekehrt ist, den Wert `1`, wenn die Strings gleich sind, den Wert `0`.

```
char *anton = "Anton";
char *zacharias = "Zacharias";
```

```
printf ("%d\n", strcmp (anton, zacharias));
printf ("%d\n", strcmp (zacharias, anton));
printf ("%d\n", strcmp (anton, anton));
```

Der Vergleich erfolgt im Sinne des verwendeten Zeichensatzes, normalerweise ASCII. Dabei kommen z. B. Großbuchstaben grundsätzlich vor den Kleinbuchstaben.

Beispielprogramm: [strings-15.c](#)

- **strcat()** – String an anderen String anhängen

Die Funktion **strcat()** hängt den zweiten String an den ersten an.

```
char *anton = "Anton";
char buffer[100] = "Huber_";
strcat (buffer, anton);
printf ("%s\n", buffer);
```

**Achtung:** Es findet keinerlei Prüfung statt, ob der resultierende String noch in den für den ersten String reservierten Speicherbereich (Puffer) hineinpaßt. Wenn man auf diese Weise über den String hinaus schreibt, werden andere Variable überschrieben, was in der Regel zu einem Absturz führt (Speicherzugriffsfehler).

Beispielprogramm: [strings-15.c](#)

- **sprintf()** – in String schreiben

**sprintf()** funktioniert ähnlich wie **printf()**, schreibt aber nicht zur Standardausgabe (Bildschirm), sondern in einen String hinein, den man als ersten Parameter übergibt.

```
char buffer[100] = "";
sprintf (buffer, "Die_Antwort_lautet:_%d", 42);
printf ("%s\n", buffer);
```

**Achtung:** Es findet keinerlei Prüfung statt, ob der Ziel-String (Puffer – *Buffer*) groß genug ist, um die Ausgabe aufzunehmen. Wenn dies nicht der Fall ist und man über das Ende des Strings hinaus schreibt, werden andere Variable des Programms überschrieben (*Buffer Overflow*), was in der Regel zu einem Absturz führt (Speicherzugriffsfehler). Derartige Fehler sind schwer zu finden und befinden sich zum Teil bis heute in Programmen, die im Internet zum Einsatz kommen und Angreifern ermöglichen, Rechner von außen zu übernehmen.

Um dieses Problem zu vermeiden, empfiehlt es sich, anstelle von **sprintf()** die Funktion **snprintf()** zu verwenden. Diese erwartet als zweiten Parameter die Länge des Ziel-Strings und sorgt dafür, daß nicht über dessen Ende hinausgeschrieben wird.

Beispielprogramm: [strings-16.c](#)

- **strstr()** – in String suchen

Die Funktion **strstr()** sucht im ersten String-Parameter nach dem zweiten und gibt als Ergebnis einen Zeiger auf diejenige Stelle zurück, an der der zweite String gefunden wurde.

```
char *answer = strstr (buffer, "Antwort");
printf ("%s\n", answer);
printf ("found_at:_%zd\n", answer – buffer);
```

Wenn man dies in einen Array-Index umrechnen will, geschieht dies durch Subtrahieren des Zeigers auf den ersten String. Das Ergebnis ist eine Integer vom Typ **ssize\_t** (*signed size type*). Um diese mit **printf()** auszugeben, verwendet man **%zd** anstelle von **%d**.

Beispielprogramm: [strings-16.c](#)

## 2.13 Parameter des Hauptprogramms

Bisher haben wir das Hauptprogramm `main()` immer in der Form

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

geschrieben.

Tatsächlich kann das Hauptprogramm vom Betriebssystem Parameter entgegennehmen (Datei: [params-1.c](#)):

```
#include <stdio.h>

int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; i < argc; i++)
        printf ("argv[%d]=\n\"%s\"", i, argv[i]);
    return 0;
}
```

Bei der ganzen Zahl `int argc` handelt es sich um die Anzahl der übergebenen Parameter.

`char **argv` ist ein Zeiger auf einen Zeiger auf `chars`, also ein Array von Arrays von `chars`, also ein Array von Strings. Wenn wir es mit einem Index `i` versehen, greifen wir auf den `i`-ten Parameter zu. Der Index `i` läuft, wie in C üblich, von 0 bis `argc - 1`. Das o. a. Beispielprogramm gibt alle übergebenen Parameter auf dem Standardausgabegerät aus:

```
$ gcc -std=c99 -Wall -O params-1.c -o params-1
$ ./params-1 foo bar baz
argc = 4
argv[0] = "./params-1"
argv[1] = "foo"
argv[2] = "bar"
argv[3] = "baz"
```

Genaugenommen übergibt das Betriebssystem dem Programm die gesamte Kommandozeile: Der nullte Parameter ist der Aufruf der ausführbaren Datei selbst – in genau der Weise, in der er eingegeben wurde.

Neben `argc` gibt es noch einen weiteren Mechanismus, mit dem das Betriebssystem dem Programm die Anzahl der übergebenen Parameter mitteilt: Als Markierung für das Ende der Liste wird ein zusätzlicher Zeiger übergeben, der auf „nichts“ zeigt, dargestellt durch die Speicheradresse mit dem Zahlenwert 0, in C mit `NULL` bezeichnet.

Um die Parameter des Programms in einer Schleife durchzugehen, können wir also entweder von 0 bis `argc - 1` zählen (Schleifenbedingung `i < argc`, Datei: [params-1.c](#) – siehe oben) oder die Schleife mit dem Erreichen der Endmarkierung abbrechen (Schleifenbedingung `argv[i] != NULL`, Datei: [params-2.c](#)).

```
#include <stdio.h>

int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; argv[i] != NULL; i++)
        printf ("argv[%d]=\n\"%s\"", i, argv[i]);
    return 0;
}
```

Auch für Zeiger gilt: `NULL` entspricht dem Wahrheitswert „falsch“; alles andere dem Wahrheitswert „wahr“. Wir dürfen die Schleifenbedingung also wie folgt abkürzen (Datei: [params-3.c](#)):

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

```

int main (int argc, char **argv)
{
    printf ("argc = %d\n", argc);
    for (char **p = argv; *p; p++)
        printf ("argv[p] = \"%s\"\n", *p);
    return 0;
}

```

## 2.14 Strukturen

In vielen Situationen ist es sinnvoll, mehrere Variable zu einer Einheit zusammenzufassen.

Das folgende Beispielprogramm `structs-1.c` faßt drei Variable `day`, `month` und `year` zu einem einzigen – neuen – Datentyp `date` zusammen:

```

#include <stdio.h>

typedef struct
{
    char day, month;
    int year;
}
date;

int main (void)
{
    date today = { 1, 11, 2016 };
    printf ("%d.%d.%d\n", today.day, today.month, today.year);
    return 0;
}

```

neuer Datentyp: date

Variable deklarieren und initialisieren

Zugriff auf die Komponente day der strukturierten Variablen today

(Zur Erinnerung: Der Datentyp `char` steht für Zahlen, die mindestens die Werte von –128 bis 127 annehmen können. C unterscheidet nicht zwischen Zahlen und darstellbaren Zeichen.)

Eine wichtige Anwendung derartiger *strukturierter Datentypen* besteht darin, zusammengehörige Daten als Einheit an Funktionen übergeben zu können (Beispielprogramm: `structs-2.c`):

```

#include <stdio.h>

typedef struct
{
    char day, month;
    int year;
}
date;

void set_date (date *d)
{
    (*d).day = 1;
    (*d).month = 11;
    (*d).year = 2016;
}

int main (void)
{
    date today;
    set_date (&today);
    printf ("%d.%d.%d\n", today.day,
            today.month, today.year);
    return 0;
}

```

Die Funktion `set_date()` hat die Aufgabe, eine `date`-Variable mit Werten zu füllen (sog. *Setter*-Funktion). Damit dies funktionieren kann, übergibt das Hauptprogramm an die Funktion einen Zeiger auf die strukturierte Variable. Über diesen Zeiger kann die Funktion dann auf alle Komponenten der Struktur zugreifen. (Die Alternative wäre gewesen, für jede Komponente einen separaten Zeiger zu übergeben.)

Da die Zeigerdereferenzierung `*foo` mit anschließendem Komponentenzugriff `(*foo).bar` eine sehr häufige Kombination ist, kennt C hierfür eine Abkürzung: `foo->bar`

Beispielprogramm: `structs-3.c`

```
#include <stdio.h>

typedef struct
{
    char day, month;
    int year;
}
date;

void set_date (date *d)
{
    d->day = 1;
    d->month = 11;
    d->year = 2016;
}

int main (void)
{
    date today;
    set_date (&today);
    printf ("%d.%d.%d\n", today.day,
            today.month, today.year);
    return 0;
}
```

## Aufgabe

Schreiben Sie eine Funktion `inc_date (date *d)` die ein gegebenes Datum `d` unter Beachtung von Schaltjahren auf den nächsten Tag setzt.

## Lösung

Wir lösen die Aufgabe über den sog. *Top-Down-Ansatz* („vom Allgemeinen zum Konkreten“). Als besonderen Trick approximieren wir unfertige Programmteile zunächst durch einfachere, fehlerbehaftete. Diese fehlerhaften Programmteile sind in den untenstehenden Beispielen rot markiert. (In der Praxis würde man diese Zeilen unmittelbar durch die richtigen ersetzen; die fehlerhaften „Platzhalter“ sollten also jeweils nur für Sekundenbruchteile im Programm stehen. Falls man einmal tatsächlich einen Platzhalter für mehrere Sekunden oder länger stehen lassen sollte – z. B., weil an mehreren Stellen Änderungen notwendig sind –, sollte man ihn durch etwas Uncompilierbares (z. B. `@@@`) markieren, damit man auf jeden Fall vermeidet, ein fehlerhaftes Programm auszuliefern.)

Zunächst kopieren wir das Beispielprogramm `structs-3.c` und ergänzen den Aufruf der – noch nicht existierenden – Funktion `inc_date()` (Datei: `:`):

```
#include <stdio.h>

typedef struct
{
    char day, month;
    int year;
```

```

}
date;

void set_date (date *d)
{
    d->day = 31;
    d->month = 1;
    d->year = 2012;
}

int main (void)
{
    date today;
    set_date (&today);
    inc_date (&today);
    printf ("%d.%d.%d\n", today.day, today.month, today.year);
    return 0;
}

```

Als nächstes kopieren wir innerhalb des Programms die Funktion `get_date()` als „Schablone“ für `inc_date()`:

```

void get_date (date *d)
{
    d->day = 31;
    d->month = 1;
    d->year = 2012;
}

void inc_date (date *d)
{
    d->day = 31;
    d->month = 1;
    d->year = 2012;
}

```

Da die Funktion jetzt existiert, ist der Aufruf nicht mehr fehlerhaft. Stattdessen haben wir jetzt eine fehlerhafte Funktion `inc_date()`.

Im nächsten Schritt ersetzen wir die fehlerhafte Funktion durch ein simples Hochzählen der `day`-Komponente (Datei: `incdate-1.c`)

```

void inc_date (date *d)
{
    d->day += 1; /* FIXME */
}

```

Diese naive Vorgehensweise versagt, sobald wir den Tag über das Ende des Monats hinauszählen. Dies reparieren wir im nächsten Schritt, wobei wir für den Moment inkorrekterweise annehmen, daß alle Monate 30 Tage hätten und das Jahr beliebig viele Monate. (Datei: `incdate-2.c`):

```

void inc_date (date *d)
{
    d->day++;
    if (d->day > 31) /* FIXME */
    {
        d->month++; /* FIXME */
        d->day = 1;
    }
}

```

Zunächst reparieren wir den Fehler, der am Ende des Jahres entsteht (Datei: `incdate-3.c`).

```

void inc_date (date *d)
{
    d->day++;
    if (d->day > 31) /* FIXME */
    {
        d->month++;
        d->day = 1;
        if (d->month > 12)
        {
            d->year++;
            d->month = 1;
        }
    }
}

```

Das Problem der unterschiedlich langen Monate gehen wir wieder stufenweise an. Zunächst ersetzen wir die Konstante 31 durch eine Variable `days_in_month`. (Datei: [incdate-4.c](#))

```

void inc_date (date *d)
{
    d->day++;
    int days_in_month = 31; /* FIXME */
    if (d->day > days_in_month)
    {
        d->month++;
        d->day = 1;
        if (d->month > 12)
        {
            d->year++;
            d->month = 1;
        }
    }
}

```

Anschließend reparieren wir den fehlerhaften (konstanten) Wert der Variablen, wobei wir zunächst das Problem der Schaltjahre aussparen (Datei: [incdate-5.c](#)):

```

void inc_date (date *d)
{
    d->day++;
    int days_in_month = 31;
    if (d->month == 2)
        days_in_month = 28; /* FIXME */
    else if (d->month == 4 || d->month == 6 || d->month == 9 || d->month == 11)
        days_in_month = 30;
    if (d->day > days_in_month)
    {
        d->month++;
        d->day = 1;
        if (d->month > 12)
        {
            d->year++;
            d->month = 1;
        }
    }
}

```

Auf dieselbe Weise lagern wir das Problem „Schaltjahr oder nicht?“ in eine Variable aus. Diese ist wieder zunächst konstant (Datei: [incdate-6.c](#)):

```

void inc_date (date *d)
{
    d->day++;

```

```

int days_in_month = 31;
if (d->month == 2)
{
    int is_leap_year = 1; /* FIXME */
    if (is_leap_year)
        days_in_month = 29;
    else
        days_in_month = 28;
}
else if (d->month == 4 || d->month == 6 || d->month == 9 || d->month == 11)
    days_in_month = 30;
if (d->day > days_in_month)
{
    d->month++;
    d->day = 1;
    if (d->month > 12)
    {
        d->year++;
        d->month = 1;
    }
}
}

```

Als nächstes ergänzen wir die Vier-Jahres-Regel für Schaltjahre (Datei [incdate-7.c](#)):

```

int is_leap_year = 0;
if (d->year % 4 == 0)
    is_leap_year = 1; /* FIXME */
if (is_leap_year)
    days_in_month = 29;
else
    days_in_month = 28;

```

Das nun vorliegende Programm arbeitet bereits für den julianischen Kalender sowie für alle Jahre von 1901 bis 2099 korrekt, nicht jedoch für z. B. das Jahr 2100 (Datei: [incdate-8.c](#)). Damit das Programm für den aktuell verwendeten gregorianischen Kalender korrekt arbeitet, ergänzen wir noch die Ausnahme, daß durch 100 teilbare Jahre keine Schaltjahre sind, sowie die Ausnahme von der Ausnahme, daß durch 400 teilbare Jahre (z. B. das Jahr 2000) eben doch Schaltjahre sind (Datei: [incdate-9.c](#)):

```

int is_leap_year = 0;
if (d->year % 4 == 0)
{
    is_leap_year = 1;
    if (d->year % 100 == 0)
    {
        is_leap_year = 0;
        if (d->year % 400 == 0)
            is_leap_year = 1;
    }
}
if (is_leap_year)
    days_in_month = 29;
else
    days_in_month = 28;

```

Damit ist die Aufgabe gelöst. Der vollständige Quelltext der Lösung (Datei: [incdate-9.c](#)) lautet:

```

#include <stdio.h>

typedef struct
{
    char day, month;
    int year;

```



```

}
date;

void set_date (date *d)
{
    d->day = 28;
    d->month = 2;
    d->year = 2000;
}

void inc_date (date *d)
{
    d->day++;
    int days_in_month = 31;
    if (d->month == 2)
    {
        int is_leap_year = 0;
        if (d->year % 4 == 0)
        {
            is_leap_year = 1;
            if (d->year % 100 == 0)
            {
                is_leap_year = 0;
                if (d->year % 400 == 0)
                    is_leap_year = 1;
            }
        }
        if (is_leap_year)
            days_in_month = 29;
        else
            days_in_month = 28;
    }
    else if (d->month == 4 || d->month == 6 || d->month == 9 || d->month == 11)
        days_in_month = 30;
    if (d->day > days_in_month)
    {
        d->month++;
        d->day = 1;
        if (d->month > 12)
        {
            d->year++;
            d->month = 1;
        }
    }
}

int main (void)
{
    date today;
    set_date (&today);
    inc_date (&today);
    printf ("%d.%d.%d\n", today.day, today.month, today.year);
    return 0;
}

```

Bemerkungen:

- Anstatt die Anzahl der Tage in einem Monat innerhalb der Funktion `set_date()` zu berechnen, ist es sinnvoll, hierfür eine eigene Funktion zu schreiben. Dasselbe gilt für die Berechnung, ob es sich bei einem gegebenem Jahr um ein Schaltjahr handelt.

- Der Top-Down-Ansatz ist eine bewährte Methode, um eine zunächst komplexe Aufgabe in handhabbare Teilaufgaben zu zerlegen. Dies hilft ungemein, in längeren Programmen (mehrere Zehntausend bis Millionen Zeilen) die Übersicht zu behalten.
- Der Trick mit dem zunächst fehlerhaften Code hat den Vorteil, daß man jeden Zwischenstand des Programms compilieren und somit austesten kann. Er birgt andererseits die Gefahr in sich, die Übersicht über den fehlerhaften Code zu verlieren, so daß es dieser bis in die Endfassung schafft. Neben dem bereits erwähnten Trick uncompilerbarer Symbole haben sich hier Kommentare wie `/*FIXME*/` bewährt, auf die man seinen Code vor der Auslieferung der Endfassung noch einmal automatisch durchsuchen läßt.

## 2.15 Dateien und Fehlerbehandlung

Die einfachste Weise, in C mit Dateien umzugehen, ist über sog. *Streams*.

Die Funktion `fopen()` erwartet als Parameter einen Dateinamen und einen Modus und gibt als Rückgabewert einen Zeiger auf einen Stream – eine Struktur vom Typ `FILE` – zurück:

```
FILE *f = fopen ("fhello.txt", "w");
```

Als Modus übergibt man eine String-Konstante. Diese kann die Buchstaben `r` für Lesezugriff (*read*), `w` für Schreibzugriff mit Überschreiben (*write*) sowie `a` für Schreibzugriff mit Anhängen (*append*) enthalten und zusätzlich den Buchstaben `b` für Binärdaten (im Gegensatz zu Text).

Die in C üblichen Ein-/Ausgabefunktionen wie z. B. `printf()` haben Varianten mit vorangestelltem „f-“, z. B. `fprintf()`. Wenn man diesen Funktionen als ersten Parameter einen Zeiger auf ein `FILE` übergibt, verhalten sie sich in der üblichen Weise, nur daß sie nicht zur Standardausgabe schreiben (Bildschirm), sondern in die Datei, deren Name beim Öffnen des `FILE` angegeben wurde (Datei `fhello-1.c`):

```
#include <stdio.h>

int main (void)
{
    FILE *f = fopen ("fhello.txt", "w");
    fprintf (f, "Hello,_world!\n");
    fclose (f);
    return 0;
}
```

Der von `fopen()` zurückgegebene Wert ist ein Zeiger. Ein Aufruf von `fprintf()` oder `fclose()` stellt eine Verwendung dieses Zeigers dar. Wenn die Datei – aus welchen Gründen auch immer – nicht geöffnet werden konnte, ist dieser Zeiger `NULL`, und seine Verwendung führt zum Absturz des Programms. Es ist daher dringend empfohlen, diesen Fall zu prüfen (Datei: `fhello-2.c`):

```
#include <stdio.h>

int main (void)
{
    FILE *f = fopen ("fhello.txt", "w");
    if (f)
    {
        fprintf (f, "Hello,_world!\n");
        fclose (f);
    }
    return 0;
}
```

Anstatt einfach nur „nichts“ zu machen, ist es besser, eine sinnvolle Fehlermeldung auszugeben. Dabei sind wir nicht allein auf „Fehler beim Öffnen der Datei“ angewiesen: Das Betriebssystem teilt uns über die globale Variable `errno` mit, was genau beim Öffnen der Datei fehlgeschlagen ist. Mit `#include <errno.h>`

erhält unser Programm Zugriff auf diese Variable und kann den Fehler-Code in seiner Fehlermeldung mit ausgeben (Datei: [fhello-3.c](#)):

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

int main (void)
{
    FILE *f = fopen ("fhello.txt", "w");
    if (f)
    {
        fprintf (f, "Hello,_world!\n");
        fclose (f);
    }
    else
        fprintf (stderr, "error_#%d\n", errno);
    return 0;
}
```

Die Ausgabe von Fehler erfolgt üblicherweise nicht mit einem „normalen“ `printf()`, sondern mit einem `fprintf()` in die bereits standardmäßig geöffnete Datei `stderr`, die *Fehlerausgabe*-Datei. Diese landet – genau wie die Standardausgabe – zunächst auf dem Bildschirm, kann aber separat von der Standardausgabe umgeleitet werden, z. B. in eine separate Datei.

Die Bedeutung der Fehler-Codes ist nicht nur in der Dokumentation des Betriebssystems, sondern auch in einer C-Bibliothek hinterlegt. Mit `#include <string.h>` erhalten wir eine Funktion `strerror()`, die den Fehler-Code in eine für Menschen lesbare Fehlermeldung umwandelt (Datei: [fhello-4.c](#)):

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

int main (void)
{
    FILE *f = fopen ("fhello.txt", "w");
    if (f)
    {
        fprintf (f, "Hello,_world!\n");
        fclose (f);
    }
    else
    {
        char *msg = strerror (errno);
        fprintf (stderr, "%s\n", msg);
    }
    return 0;
}
```

Ein häufiger Fall ist, daß das Programm nach Ausgabe der Fehlermeldung direkt beendet werden soll. Hierbei wird nicht das sonst übliche `return 0` des Hauptprogramms aufgerufen, sondern `return` mit einer anderen Zahl als 0, z. B. `return 1` für „allgemeiner Fehler“. Üblich ist es, den Fehler-Code zurückgegeben – `return errno` –, um diesen auch an diejenigen, der das Programm aufgerufen hat, weiterzureichen.

Für diese standardisierte Reaktion auf Fehler steht mit `#include <error.h>` eine Funktion `error()` zur Verfügung, die eine zum übergebenen Fehler-Code gehörende Fehlermeldung ausgibt und anschließend das Programm mit einem übergebenen Fehler-Code beendet (Datei: [fhello-5.c](#)):

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

```

int main (void)
{
    FILE *f = fopen ("fhello.txt", "w");
    if (!f)
        error (1, errno, "cannot_open_file");
    fprintf (f, "Hello,_world!\n");
    fclose (f);
    return 0;
}

```

In diesem Fall ist `1` der Code, den das Programm im Fehlerfall zurückgeben soll, und `errno` ist die Nummer des Fehlers, dessen Fehlermeldung auf dem Bildschirm (`stderr`) ausgegeben werden soll. (Üblich wäre wie gesagt auch, hier zweimal `errno` zu übergeben.)

**Bitte niemals Fehler einfach ignorieren!** Ein Programm, das bereits auf eine nicht gefundene Datei mit einem Absturz reagiert, ist der Alptraum jedes Benutzers und eines jeden, der versucht, in dem Programm Fehler zu beheben. Ein korrekt geschriebenes Programm stürzt *niemals* ab, sondern beendet sich schlimmstensfalls mit einer aussagekräftigen Fehlermeldung, die uns in die Lage versetzt, die Fehlerursache zu beheben.

## 3 Bibliotheken

### 3.1 Der Präprozessor

Der erste Schritt beim Compilieren eines C-Programms ist das Auflösen der sogenannten Präprozessor-Direktiven und -Macros.

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

bewirkt, daß aus Sicht des Compilers anstelle der Zeile der Inhalt der Datei `stdio.h` im C-Quelltext erscheint. Dies ist zunächst unabhängig von Bibliotheken und auch nicht auf die Programmiersprache C beschränkt.

Beispiel: Die Datei `maerchen.c` enthält:

```

Vor langer, langer Zeit
gab es einmal
#include "hexe.h"
Die lebte in einem Wald.

```

Die Datei `hexe.h` enthält:

```
eine kleine Hexe.
```

Der Aufruf

```
$ gcc -E -P maerchen.c
```

produziert die Ausgabe

```

Vor langer, langer Zeit
gab es einmal
eine kleine Hexe.
Die lebte in einem Wald.

```

Mit der Option `-E` weisen wir `gcc` an, nicht zu compilieren, sondern nur den Präprozessor aufzurufen. Die Option `-P` unterdrückt Herkunftsangaben, die normalerweise vom Compiler verwendet werden, um Fehlermeldungen den richtigen Zeilen in den richtigen Dateien zuordnen zu können. Ohne das `-P` lautet die Ausgabe:

```

# 1 "maerchen.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "maerchen.c"

```

```

Vor langer, langer Zeit
gab es einmal
# 1 "hexe.h" 1
eine kleine Hexe.
# 3 "maerchen.c" 2
Die lebte in einem Wald.

```

Nichts anderes geschieht, wenn man das klassische `hello.c` (Datei: `hello-1.c` compiliert:

```

#include <stdio.h>

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

```

Die Datei `stdio.h` ist wesentlich länger als `hexe.txt` in dem o. a. Beispiel, und sie ruft weitere Include-Dateien auf, so daß wir insgesamt auf über 800 Zeilen Quelltext kommen.

Die spitzen Klammern anstelle der Anführungszeichen bedeuten, daß es sich um eine *Standard-Include-Datei* handelt, die nur in den Standard-Include-Verzeichnissen gesucht werden soll, nicht jedoch im aktuellen Verzeichnis.

## 3.2 Bibliotheken einbinden

Tatsächlich ist von den über 800 Zeilen aus `stdio.h` nur eine einzige relevant, nämlich:

```
extern int printf (__const char *__restrict __format, ...);
```

Dies ist die Deklaration einer Funktion, die sich von einer „normalen“ Funktionsdefinition nur wie folgt unterscheidet:

- Die Parameter `__const char *__restrict __format, ...` heißen etwas ungewöhnlich.
- Der Funktionskörper `{ ... }` fehlt. Stattdessen folgt auf die Kopfzeile direkt ein Semikolon `;`.
- Der Deklaration ist das Schlüsselwort `extern` vorangestellt.

Dies bedeutet für den Compiler: „Es gibt diese Funktion und sie sieht aus, wie beschrieben. Benutze sie einfach, und kümmere dich nicht darum, wer die Funktion schreibt.“

Wenn wir tatsächlich nur `printf()` benötigen, können wir also die Standard-Datei `stdio.h` durch eine eigene ersetzen, die nur die o. a. Zeile `extern int printf (...)` enthält. (Dies ist in der Praxis natürlich keine gute Idee, weil nur derjenige, der die Funktion `printf()` geschrieben hat, den korrekten Aufruf kennt. In der Praxis sollten wir immer diejenige Include-Datei benutzen, die gemeinsam mit der tatsächlichen Funktion ausgeliefert wurde.)

Der Präprozessor kann nicht nur `#include`-Direktiven auflösen. Mit `#define` kann man sog. *Makros* definieren, die bei Benutzung durch einen Text ersetzt werden. Auf diese Weise kann man *Konstante* definieren.

Beispiel: `higher-math-1.c`

```

#include <stdio.h>

#define VIER 4

int main (void)
{
    printf ("2_+_2_=_%d\n", VIER);
    return 0;
}

```

Genau wie bei **#include** nimmt der Präprozessor auch bei **#define** eine rein textuelle Ersetzung vor, ohne sich um den Sinn des Ersetzten zu kümmern.

Beispiel: [higher-math-2.c](#)

```
#include <stdio.h>

#define wuppdich printf
#define holla main
#define pruzzel return
#define VIER 4

int holla (void)
{
    wuppdich ("2_+_2_=_%d\n", VIER);
    pruzzel 0;
}
```

Dies kann zu Problemen führen, sobald Berechnungen ins Spiel kommen.

Beispiel: [higher-math-3.c](#)

```
#include <stdio.h>

#define VIER 2 + 2

int main (void)
{
    printf ("2_+_3_*_4_=_%d\n", 2 + 3 * VIER);
    return 0;
}
```

Hier z. B. sieht man mit `gcc -E rechnen.c`, daß die Ersetzung des Makros **VIER** wie folgt lautet:

```
printf ("2_+_3_*_4_=_%d\n", 2 + 3 * 2 + 2);
```

Der C-Compiler wendet die Regel „Punktrechnung geht vor Strichrechnung“ an und erfährt überhaupt nicht, daß das `2 + 2` aus einem Makro entstanden ist.

Um derartige Effekte zu vermeiden, setzt man arithmetische Operationen innerhalb von Makros in Klammern.

Beispiel: [higher-math-4.c](#)

```
#define VIER (2 + 2)
```

(Es ist in den meisten Situationen üblich, Makros in **GROSSBUCHSTABEN** zu benennen, um darauf hinzuweisen, daß es sich eben um einen Makro handelt.)

Achtung: Hinter Makro-Deklaration kommt üblicherweise *kein* Semikolon.

Wenn man ein Semikolon setzt, gehört dies mit zum Ersetzungstext des Makros. Dies ist grundsätzlich zulässig, führt aber zu sehr seltsamen C-Quelltexten. – siehe z. B. [higher-math-5.c](#).

Das nächste Beispiel illustriert, wie man Bibliotheken schreibt und verwendet.

Es besteht aus drei Quelltexten:

[philosophy.c](#):

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

int main (void)
{
    printf ("The_answer_is_%d.\n", answer ());
    return 0;
}
```

answer.c:

```
int answer(void)
{
    return 42;
}
```

answer.h:

```
extern int answer(void);
```

Das Programm `philosophy.c` verwendet eine Funktion `answer()`, die in der Datei `answer.h` extern deklariert ist.

Der „normale“ Aufruf

```
$ gcc -Wall -O philosophy.c -o philosophy
```

liefert die Fehlermeldung:

```
/tmp/ccr4Njg7.o: In function 'main':
philosophy.c:(.text+0xa): undefined reference to 'answer'
collect2: ld returned 1 exit status
```

Diese stammt nicht vom Compiler, sondern vom *Linker*. Das Programm ist syntaktisch korrekt und wird auch korrekt in eine Binärdatei umgewandelt (hier: `/tmp/ccr4Njg7.o`). Erst beim Zusammenbau („Linken“) der ausführbaren Datei (`philosophy`) tritt ein Fehler auf.

Tatsächlich wird die Funktion `answer()` nicht innerhalb von `philosophy.c`, sondern in einer separaten Datei `answer.c`, einer sog. *Bibliothek* definiert. Es ist möglich (und üblich), Bibliotheken separat vom Hauptprogramm zu compilieren. Dadurch spart man sich das Neucompilieren, wenn im Hauptprogramm etwas geändert wurde, nicht jedoch in der Bibliothek.

Mit der Option `-c` weisen wir `gcc` an, nur zu compilieren, jedoch nicht zu linkern. Die Aufrufe

```
$ gcc -Wall -O -c philosophy.c
$ gcc -Wall -O -c answer.c
```

produzieren die Binärdateien `philosophy.o` und `answer.o`, die sogenannten *Objekt-Dateien* (daher die Endung `.o` oder `.obj`).

Mit dem Aufruf

```
$ gcc philosophy.o answer.o -o philosophy
```

bauen wir die beiden bereits compilierten Objekt-Dateien zu einer ausführbaren Datei `philosophy` (hier ohne Endung) zusammen.

Es ist auch möglich, im Compiler-Aufruf gleich beide C-Quelltexte zu übergeben:

```
$ gcc -Wall -O philosophy.c answer.c -o philosophy
```

In diesem Fall ruft `gcc` zweimal den Compiler auf (für jede C-Datei einmal) und anschließend den Linker.

### 3.3 Bibliotheken verwenden (Beispiel: OpenGL)

Die *OpenGL*-Bibliothek dient dazu, unter Verwendung von Hardware-Unterstützung dreidimensionale Grafik auszugeben.

Die einfachste Art und Weise, OpenGL in seinen Programmen einzusetzen, erfolgt über eine weitere Bibliothek, das *OpenGL Utility Toolkit (GLUT)*.

Die Verwendung von OpenGL und GLUT erfolgt durch Einbinden der Include-Dateien

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

und die Übergabe der Bibliotheken `-lGL -lGLU -lglut` im Compiler-Aufruf:

```
$ gcc -Wall -O cube.c -lGL -lGLU -lglut -o cube
```

(Dies ist der Aufruf unter Unix. Unter Microsoft Windows ist der Aufruf etwas anders und hängt von der verwendeten Version der GLUT-Bibliothek ab. Für Details siehe die Dokumentation der GLUT-Bibliothek sowie die Datei [egal.txt](#).)

Die Bibliothek stellt uns fertig geschriebene Programmfragmente zur Verfügung, insbesondere:

- Funktionen: z. B. `glutInit (&argc, argv);`
- Konstante: z. B. `GLUT_RGBA`
- Datentypen: z. B. `GLfloat`

Manche OpenGL-Funktionen erwarten als Parameter ein Array. Dies gilt z. B. beim Setzen von Farben oder beim Positionieren einer Lichtquelle:

```
GLfloat light0_position[] = {1.0, 2.0, -2.0, 1.0};  
glLightfv (GL_LIGHT0, GL_POSITION, light0_position);
```

Ein weiteres wichtiges allgemeines Konzept, das in OpenGL eine Rolle spielt, ist die Übergabe einer Funktion an die Bibliothek. Man nennt dies das *Installieren einer Callback-Funktion*.

```
void draw (void)  
{ ... }
```

```
glutDisplayFunc (draw);
```

Wir übergeben die – von uns geschriebene – Funktion `draw()` an die OpenGL-Funktion `glutDisplayFunc()`. Dies bewirkt, daß OpenGL immer dann, wenn etwas gezeichnet werden soll, `draw()` aufruft. Innerhalb von `draw()` können wir also unsere Zeichenbefehle unterbringen.

Die OpenGL-Bibliothek ist sehr umfangreich und kann im Rahmen dieser Vorlesung nicht im Detail behandelt werden. Um trotzdem damit arbeiten zu können, lagern wir bestimmte Teile – Initialisierung und das Setzen von Farben – in eine eigene Bibliothek `opengl-magic` aus, die wir als „Black Box“ verwenden. Der Compiler-Aufruf lautet dann:

```
$ gcc -Wall -O cube.c -lGL -lGLU -lglut opengl-magic.c -o cube
```

(Wer in eigenen Projekten mehr mit OpenGL machen möchte, ist herzlich eingeladen, die Funktionsweise von `opengl-magic` zu studieren.)

- Das Beispielprogramm `cube-1.c` illustriert, wie man grundsätzlich überhaupt ein geometrisches Objekt zeichnet. In diesem Fall handelt es sich um einen Würfel der Kantenlänge `0.5`, von dem wir nur die Vorderfläche sehen, also ein Quadrat.

```
#include <GL/gl.h>  
#include <GL/glu.h>  
#include <GL/glut.h>  
#include "opengl-magic.h"  
  
void draw (void)  
{  
    glClear (GL_COLOR_BUFFER_BIT + GL_DEPTH_BUFFER_BIT);  
    set_material_color (1.0, 0.7, 0.0);  
    glutSolidCube (0.75);  
    glFlush ();  
    glutSwapBuffers ();  
}  
  
int main (int argc, char **argv)  
{  
    init_opengl (&argc, argv, "Cube");  
    glutDisplayFunc (draw);  
    glutMainLoop ();  
    return 0;  
}
```



- In `cube-2.c` kommt eine Drehung um  $-30$  Grad um eine schräge Achse  $(0.5, 1.0, 0.0)$  hinzu. Der Würfel ist jetzt als solcher zu erkennen.

Jeder Aufruf von `glRotatef()` bewirkt, daß alle nachfolgenden Zeichenoperationen gedreht ausgeführt werden.

- In `cube-3.c` kommt als zusätzliches Konzept eine weitere Callback-Funktion hinzu, nämlich ein *Timer-Handler*. Durch den `glutTimerFunc()`-Aufruf veranlassen wir OpenGL, die von uns geschriebene Funktion `timer_handler()` aufzurufen, sobald 50 Millisekunden vergangen sind.

Innerhalb von `timer_handler()` rufen wir `glutTimerFunc()` erneut auf, was insgesamt zur Folge hat, daß `timer_handler()` periodisch alle 50 Millisekunden aufgerufen wird.

Die „Nutzlast“ der Funktion `timer_handler()` besteht darin, eine Variable `t` um den Wert `0.05` zu erhöhen und anschließend mittels `glutPostRedisplay()` ein Neuzeichnen anzufordern. Dies alles bewirkt, daß die Variable `t` die aktuelle Zeit seit Programmbeginn in Sekunden enthält und daß `draw()` zwanzigmal pro Sekunde aufgerufen wird.

- In `cube-3.c` dreht sich der Würfel zunächst langsam, dann immer schneller. Dies liegt daran, daß sich jedes `glRotatef()` auf alle nachfolgenden Zeichenbefehle auswirkt, so daß sich sämtliche `glRotatef()` aufaddieren.

Eine Möglichkeit, stattdessen eine gleichmäßige Drehung zu erreichen, besteht darin, den Wirkungsbereich des `glRotatef()` zu begrenzen. Dies geschieht durch Einschließen der Rotation in das Befehlspaar `glPushMatrix()` und `glPopMatrix()`: Durch `glPopMatrix()` wird das System wieder in denjenigen Zustand versetzt, in dem es sich vor dem Aufruf von `glPushMatrix()` befand.

Dies ist in `cube-4.c` (langsame Drehung) und `cube-5.c` (schnelle Drehung) umgesetzt.

## Aufgabe

Für welche elementaren geometrischen Körper stellt die GLUT-Bibliothek Zeichenroutinen zur Verfügung?

## Lösung

Ein Blick in die Include-Datei `glut.h` verweist uns auf eine andere Include-Datei:

```
#include "freeglut_std.h"
```

Wenn wir darin nach dem Wort `glutSolidCube` suchen, finden wir die Funktionen:

```
glutSolidCube (GLdouble size);
glutSolidSphere (GLdouble radius, GLint slices, GLint stacks);
glutSolidCone (GLdouble base, GLdouble height, GLint slices, GLint stacks);
glutSolidTorus (GLdouble innerRadius, GLdouble outerRadius, GLint sides, GLint rings);
glutSolidDodecahedron (void);
glutSolidOctahedron (void);
glutSolidTetrahedron (void);
glutSolidIcosahedron (void);
glutSolidTeapot (GLdouble size);
```

Zu jeder `glutSolid`-Funktion gibt es auch eine `glutWire`-Funktion, beispielsweise `glutWireCube()` als Gegenstück zu `glutSolidCube()`.

In demselben Verzeichnis finden wir auch eine Datei `freeglut_ext.h` mit weiteren Funktionen dieses Typs:

```
glutSolidRhombicDodecahedron (void);
glutSolidSierpinskiSponge (int num_levels, GLdouble offset[3], GLdouble scale);
glutSolidCylinder (GLdouble radius, GLdouble height, GLint slices, GLint stacks);
```

Die GLUT-Bibliothek kennt insbesondere standardmäßig eine Funktion zum Zeichnen einer Teekanne und als Erweiterung eine Funktion zum Zeichnen eines Sierpinski-Schwamms.

Die weiteren OpenGL-Beispielprogramme illustrieren den Umgang mit Transformationen.

- Die Beispielprogramme [cube-5.c](#) und [cube-6.c](#) illustrieren eine weitere Transformation der gezeichneten Objekte, nämlich die Translation (Verschiebung).

Jeder Transformationsbefehl wirkt sich jeweils auf die *danach* erfolgenden Zeichenbefehle aus. Um sich zu veranschaulichen, welche Transformationen auf ein gezeichnetes Objekt wirken (hier z. B. auf [glutSolidCube\(\)](#)), muß man die Transformationen in der Reihenfolge *von unten nach oben* ausführen.

- Das Beispielprogramm [orbit-1.c](#) verwendet weitere Transformationen und geometrische Objekte, um die Umlaufbahn des Mondes um die Erde zu illustrieren.

Darüberhinaus versteht es die gezeichneten Objekte „Mond“ und „Erde“ mit realistischen Texturen (NASA-Fotos). Die hierfür notwendigen doch eher komplizierten Funktionsaufrufe wurden wiederum in eine Bibliothek ([textured-spheres](#)) ausgelagert.

### 3.4 Projekt organisieren: make

In größeren Projekten ruft man den Compiler (und Präprozessor und Linker) nicht „von Hand“ auf, sondern überläßt dies einem weiteren Programm namens [make](#).

[make](#) sucht im aktuellen Verzeichnis nach einer Datei [Makefile](#) (ohne Dateiendung). (Normalerweise gibt es nur ein Makefile pro Verzeichnis. Falls es doch mehrere gibt, kann man die Datei, z. B. [Makefile.1](#), mit [-f](#) auch explizit angeben: [make -f Makefile.1](#).)

#### 3.4.1 make-Regeln

Ein Makefile enthält sog. Regeln, um Ziele zu erzeugen. Eine Regel beginnt mit der Angabe des Ziels, gefolgt von einem Doppelpunkt und den Dateien (oder anderen Zielen), von denen es abhängt. Darunter steht, mit einem Tabulator-Zeichen eingerückt, der Programmaufruf, der nötig ist, um das Ziel zu bauen.

```
philosophy.o: philosophy.c answer.h
    gcc -c philosophy.c -o philosophy.o
```

Achtung: Ein Tabulator-Zeichen läßt sich optisch häufig nicht von mehreren Leerzeichen unterscheiden. [make](#) akzeptiert jedoch nur das Tabulator-Zeichen.

Die o. a. Regel bedeutet, daß jedesmal, wenn sich [philosophy.c](#) oder [answer.h](#) geändert hat, [make](#) das Programm [gcc](#) in der beschriebenen Weise aufrufen soll.

Durch die Kombination mehrerer Regeln lernt [make](#), welche Befehle in welcher Reihenfolge aufgerufen werden müssen, je nachdem, welche Dateien geändert wurden.

Beispiel: [Makefile.orbit-x1](#)

Der Aufruf

```
$ make -f Makefile.orbit-x1
```

bewirkt beim ersten Mal:

```
gcc -Wall -O orbit-x1.c opengl-magic-double.c textured-spheres.c \
    -lGL -lGLU -lglut -o orbit-x1
```

Beim zweiten Aufruf stellt [make](#) fest, daß sich keine der Dateien auf der rechten Seite der Regeln (rechts vom Doppelpunkt) geändert hat und ruft keine Programme auf:

```
make: »orbit-x1« ist bereits aktualisiert.
```

### 3.4.2 make-Macros

Um wiederkehrende Dinge (typischerweise: Listen von Dateinamen oder Compiler-Optionen) nicht mehrfach eingeben zu müssen, kennt `make` sog. Macros:

```
PHILOSOPHY_SOURCES = philosophy.c answer.h
```

Um den Macro zu expandieren, setzt man ihn in runde Klammern mit einem vorangestellten Dollarzeichen. Die Regel

```
philosophy.o: $(PHILOSOPHY_SOURCES)
    gcc -c philosophy.c -o philosophy.o
```

ist also nur eine andere Schreibweise für:

```
philosophy.o: philosophy.c answer.h
    gcc -c philosophy.c -o philosophy.o
```

Beispiel: `Makefile.blink`

Die Beispielprogramme `blink-*.c` sind dafür gedacht, auf einem Mikrocontroller zu laufen. Der Compiler-Aufruf erfordert zusätzliche Optionen (z. B. `-Os -mmcu=atmega32`), und es müssen zusätzliche Entwicklungswerkzeuge (z. B. `avr-objcopy`) aufgerufen werden – ebenfalls mit den richtigen Optionen. Der Prozeß des Zusammenbauens wird durch ein Makefile `Makefile.blink` verwaltet.

`Makefile.blink` speichert den Namen des Quelltextes (ohne die Endung `.c`) in einem Macro. Durch Ändern allein dieses Macros ist es daher möglich, das Makefile für ein anderes Projekt einzusetzen.

Zusätzlich führt `Makefile.blink` eine neue Regel `clean` ein. Diese bewirkt üblicherweise, daß alle automatisch erzeugten Dateien gelöscht werden:

```
clean:
    rm -f $(TARGET).elf $(TARGET).hex
```

Rechts vom Doppelpunkt nach `clean` befinden sich keine Abhängigkeitsdateien. Dies hat zur Folge, daß die Aktion (`rm -f ...`) bei `make clean` grundsätzlich immer ausgeführt wird, also nicht nur, wenn sich irgendeine Datei geändert hat.

Ebenfalls üblich ist eine weitere Regel `install`, die bewirkt, daß die Zielformate (bei einem Programm typischerweise eine ausführbare Datei, bei einer Bibliothek typischerweise `.a`-, `.so`- sowie `.h`-Dateien) an ihren endgültigen Bestimmungsort kopiert werden.

### 3.4.3 Fazit: 3 Sprachen

Um in C programmieren zu können, muß man also tatsächlich drei Sprachen lernen:

- C selbst,
- die Präprozessor-Sprache
- und die `make`-Sprache.

Durch Entwicklungsumgebungen wie z. B. `Eclipse` läßt sich der `make`-Anteil teilweise automatisieren. (`Eclipse` z. B. schreibt ein Makefile; andere Umgebungen übernehmen die Funktionalität von `make` selbst.) Auch dort muß man jedoch die zu einem Projekt gehörenden Dateien verwalten.

Wenn sich dann eine Datei nicht an dem Ort befindet, erhält man u. U. wenig aussagekräftige Fehlermeldungen, z. B.:

```
make: *** No rule to make target 'MeinProgramm.elf', needed by 'elf'. Stop.
```

Wenn man dann um die Zusammenhänge weiß („Welche Bibliotheken verwendet mein Programm? Wo befinden sich diese? Und wie erfährt der Linker davon?“), kann man das Problem systematisch angehen und ist nicht auf Herumraten angewiesen.

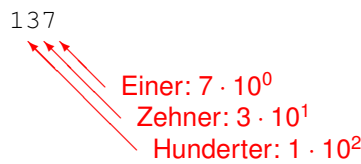
## 4 Hardwarenahe Programmierung

### 4.1 Bit-Operationen

#### 4.1.1 Zahlensysteme

##### Dezimalsystem

- Basis: 10
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

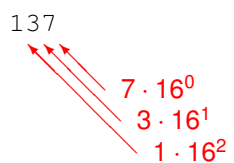


$137_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0 = 100 + 30 + 7 = 137$

Handwritten addition:  
$$\begin{array}{r} 137 \\ + 42 \\ \hline 179 \end{array}$$

##### Hexadezimalsystem

- Basis: 16
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F



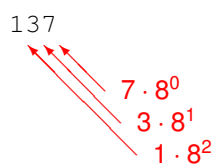
$137_{16} = 1 \cdot 16^2 + 3 \cdot 16^1 + 7 \cdot 16^0 = 256 + 48 + 7 = 311$

Handwritten addition:  
$$\begin{array}{r} A380 \\ + B747 \\ \hline 15AC7 \end{array}$$

- Schreibweise in C: `0x137`

##### Oktalsystem

- Basis: 8
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7



$137_8 = 1 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 = 64 + 24 + 7 = 95$   
 $42_8 = 4 \cdot 8^1 + 2 \cdot 8^0 = 32 + 2 = 34$   
 $201_8 = 2 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0 = 128 + 1 = 129$

Handwritten addition:  
$$\begin{array}{r} 137 \\ + 42 \\ \hline 201 \end{array}$$

- Schreibweise in C: `0137`

## Rechner für beliebige Zahlensysteme: GNU bc

```
$ bc
ibase=8
137      ← Eingabe zur Basis 8
95       ← Ausgabe zur Basis 10
obase=10 ← Eingabe zur Basis 8 (108 = 8)
137 + 42
201      ← Ausgabe zur Basis 8
```

## Binärsystem

- Basis: 2
- Gültige Ziffern: 0, 1

110

110  
+ 1100  
----  
10010

$0 \cdot 2^0$   
 $1 \cdot 2^1$   
 $1 \cdot 2^2$

$$110_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 4 + 2 + 0 = 6$$

- Binär-Zahlen ermöglichen es, elektronisch zu rechnen ...
- und mehrere „Ja/Nein“ (Bits) zu einer einzigen Zahl zusammenzufassen.
- Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:

000 0	0000 0	1000 8
001 1	0001 1	1001 9
010 2	0010 2	1010 A
011 3	0011 3	1011 B
100 4	0100 4	1100 C
101 5	0101 5	1101 D
110 6	0110 6	1110 E
111 7	0111 7	1111 F

Beispiel:  $1101011_2 = 153_8 = 6B_{16}$

Anwendungsbeispiel: Oktal-Schreibweise für Unix-Zugriffsrechte

```
-rw-r----- = 01101000002 = 6408    $ chmod 640 file.c
-rwxr-x---   = 01111010002 = 7508    $ chmod 750 subdir
```

## IP-Adressen (IPv4)

- Basis: 256
- Gültige Ziffern: 0 bis 255, getrennt durch Punkte
- Kompakte Schreibweise für Binärzahlen mit 32 Ziffern (Bits)

192.168.0.1

$1 \cdot 256^0$   
 $0 \cdot 256^1$   
 $168 \cdot 256^2$   
 $192 \cdot 256^3$

$$192.168.0.1_{256} = 11000000\ 10101000\ 00000000\ 00000001_2$$

## 4.1.2 Bit-Operationen in C

$\begin{array}{r} 0110 \\ + 1100 \\ \hline 10010 \end{array}$	$\begin{array}{r} 0110 \\   1100 \\ \hline 1110 \end{array}$	$\begin{array}{r} 0110 \\ \& 1100 \\ \hline 0100 \end{array}$	$\begin{array}{r} 0110 \\ \wedge 1100 \\ \hline 1010 \end{array}$	$\begin{array}{r} \sim 1100 \\ \hline 0011 \end{array}$	$\begin{array}{r} 0110 \\ >> 2 \\ \hline 0001 \end{array}$
Addition	Oder	Und	Exklusiv-Oder	Negation	Bit-Verschiebung

$\begin{array}{r} 01101100 \\   00000010 \\ \hline 01101110 \end{array}$	$\begin{array}{r} 01101100 \\ \& 11110111 \\ \hline 01100100 \end{array}$	$\begin{array}{r} 01101100 \\ \wedge 00010000 \\ \hline 01111100 \end{array}$
Bit gezielt setzen	Bit gezielt löschen	Bit gezielt umklappen

- Bits werden häufig von rechts und ab 0 numeriert (hier: 0 bis 7), um die Maskenerzeugung mittels Schiebeoperatoren zu erleichtern.
- Die Bit-Operatoren (z. B. `&` in C) wirken jeweils auf alle Bits der Zahlen. Die logischen Operatoren (z. B. `&&` in C) prüfen die Zahl insgesamt auf  $\neq 0$ . Nicht verwechseln!

`6 & 12 == 4`  
`6 && 12 == 1`

Anwendung: Bit 2 (also das dritte Bit von rechts) in einer 8-Bit-Zahl auf 1 setzen:

$\begin{array}{r} 00000001 \\ << 2 \\ \hline 00000100 \end{array}$	$\begin{array}{r} 01101100 \\   00000100 \\ \hline 01101100 \end{array}$
Maske für Bit 2	Bit gezielt setzen

- Schreibweise in C: `a |= 1 << 2;`

Anwendung: Bit 2 in einer 8-Bit-Zahl auf 0 setzen:

$\begin{array}{r} 00000001 \\ \ll 2 \\ \hline 00000100 \end{array}$	$\begin{array}{r} \sim 00000100 \\ \hline 11111011 \end{array}$	$\begin{array}{r} 01101100 \\ \& 11111011 \\ \hline 01101000 \end{array}$
Maske zum Löschen von Bit 2 erzeugen		Bit gezielt löschen

- Schreibweise in C: `a &= ~(1 << 2);`

Anwendung: Bit 2 aus einer 8-Bit-Zahl extrahieren:

$\begin{array}{r} 00000001 \\ << 2 \\ \hline 00000100 \end{array}$	$\begin{array}{r} 01101100 \\ \& 00000100 \\ \hline 00000100 \end{array}$	$\begin{array}{r} 00000100 \\ >> 2 \\ \hline 00000001 \end{array}$
Maske für Bit 2	Bit 2 isolieren	in Zahl 0 oder 1 umwandeln

- Schreibweise in C: `x = (a & (1 << 2)) >> 2;`

Beispiel: Netzmaske für 256 IP-Adressen

```
192.168. 1.123 ← IP-Adresse eines Rechners
& 255.255.255. 0 ← Netzmaske: 255 = 111111112
-----
192.168. 1. 0 ← IP-Adresse des Sub-Netzes
```

Beispiel: Netzmaske für 8 IP-Adressen

```
192.168. 1.123 ← IP-Adresse eines Rechners
& 255.255.255.248 ← Netzmaske
-----
192.168. 1.120 ← IP-Adresse des Sub-Netzes
```

```
01111011
& 11111000
-----
01111000
```

## 4.2 Programmierung von Mikrocontrollern

Ein Mikrocontroller ist ein elektronischer Baustein, der einen kompletten Computer mit eingeschränkter Funktionalität enthält.

Mit „eingeschränkter Funktionalität“ ist gemeint, daß auf einem Mikrocontroller kein Universal-Betriebssystem läuft, sondern daß darauf nur ein einziges Programm läuft, nämlich die Anwendungs-Software.

Wenn ein Baustein einen kompletten Computer enthält, der leistungsfähig genug ist, daß darauf ein Universal-Betriebssystem laufen kann, spricht man normalerweise nicht mehr von einem Mikrocontroller, sondern von einem Ein-Chip-Computer. Der Übergang ist fließend.

Da ein Mikrocontroller nicht über die Leistung verfügt, einen Compiler laufen zu lassen, erfolgt die Entwicklung von Software für Mikrocontroller auf einem „normalen“ Computer. Diese Art der Software-Entwicklung, bei der ein Computer Software für einen ganz anderen Computer erzeugt, bezeichnet man als *Cross-Entwicklung*. Die einzelnen Werkzeuge heißen entsprechend *Cross-Compiler*, *Cross-Assembler* und *Cross-Linker*.

Beispiel: Erzeugen einer ausführbaren Datei `blink.elf` aus einem C-Quelltext `blink.c` für einen Mikrocontroller vom Typ ATmega328P

```
avr-gcc -Wall -Os -mmcu=atmega328p blink.c -o blink.elf
```

Damit der Mikrocontroller die ausführbare Datei ausführen kann, muß man sie mit einem speziellen Werkzeug auf den Mikrocontroller *herunterladen*. Hierfür ist es oft notwendig, die Datei vorher in ein anderes Dateiformat zu konvertieren.

Beispiel: Konvertierung der ausführbaren Datei `blink.elf` aus dem ELF-Dateiformat in das Intel-HEX-Format:

```
avr-objcopy -O ihex blink.elf blink.hex
```

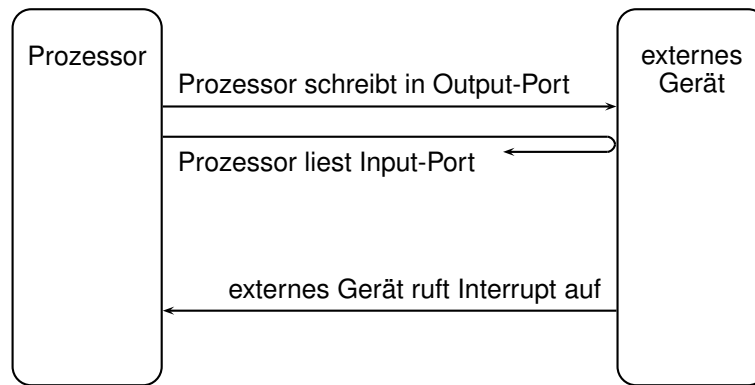
Anschließend kann die Datei auf den Mikrocontroller heruntergeladen werden. Beispiel: Herunterladen der Datei `blink.hex` in den Flash-Speicher eines ATmega328P-Mikrocontrollers auf einem Mikrocontroller-Board vom Typ Arduino Uno über die Schnittstelle `/dev/ttyACM0`

```
avrdude -P /dev/ttyACM0 -c arduino -p m328p -U flash:w:blink.hex
```

## 4.3 I/O-Ports

Es gibt drei grundlegende Mechanismen für die Kommunikation zwischen dem Prozessor und einem externen Gerät:

- Über Output-Ports kann der Prozessor das Gerät aktiv steuern,
- über Input-Ports kann er es aktiv abfragen,
- und über Interrupts kann das externe Gerät im Prozessor Aktivitäten auslösen.



Input- und Output-Ports, zusammengefaßt: I/O-Ports, sind spezielle Speicherzellen, die mit einem externen Gerät verbunden sind.

- Ein in einen Output-Port geschriebener Wert bewirkt eine Spannungsänderung in einer Leitung, die zu einem externen Gerät führt.
- Wenn ein externes Gerät eine Spannung an eine Leitung anlegt, die zu einer Speicherzelle führt, kann der Prozessor diese als Input-Port lesen.

Um z. B. auf einen Druck auf einen Taster zu warten, kann ein Program periodisch in einer Schleife einen Input-Port lesen und die Schleife erst dann beenden, wenn der Wert für „Taster gedrückt“ gelesen wurde.

Diese Methode heißt „Busy Waiting“: Der Prozessor ist vollständig mit Warten beschäftigt. Wenn gleichzeitig noch andere Aktionen stattfinden sollen, müssen diese in der Schleife mit berücksichtigt werden.

Beispiel für die Verwendung eines Output-Ports: Roboter-Steuerung

Datei: RP6Base/RP6Base\_Examples/RP6Examples\_20080915/RP6Lib/RP6base/RP6RobotBaseLib.c

Suchbegriff: setMotorDir

```

void setMotorDir(uint8_t left_dir, uint8_t right_dir)
{
    mleft_dir = left_dir;
    mright_dir = right_dir;
    mleft_des_dir = left_dir;
    mright_des_dir = right_dir;
    if(left_dir)
        PORTC |= DIR_L;
    else
        PORTC &= ~DIR_L;
    if(right_dir)
        PORTC |= DIR_R;
    else
        PORTC &= ~DIR_R;
}
  
```

Die Variable `PORTC` ist ein Output-Port. Durch Manipulation einzelner Bits in dieser Variablen ändert sich die Spannung an den elektrischen „Beinchen“ des Mikrocontrollers. Hierdurch wird die Beschaltung von Elektromotoren umgepolt.

(Die Konstanten `DIR_L` und `DIR_R` sind „Bitmasken“, d. h. Zahlen, die in ihrer Binärdarstellung nur eine einzige 1 und ansonsten Nullen haben. Durch die Oder- und Und-Nicht-Operationen werden einzelne Bits in `PORTC` auf 1 bzw. 0 gesetzt.)

Die direkte Ansteuerung von I/O-Ports ist nur auf Mikrocontrollern üblich. Auf Personal-Computern erfolgt die gesamte Ein- und Ausgabe über Betriebssystem-„Treiber“. Anwenderprogramme greifen dort i. d. R. nicht direkt auf I/O-Ports zu.



## 4.4 Interrupts

Ein Interrupt ist ein Unterprogramm, das nicht durch einen Befehl ([call](#)), sondern durch ein externes Gerät (über ein Stromsignal) aufgerufen wird.

Damit dies funktioniert, muß die Adresse, an der sich das Unterprogramm befindet, an einer jederzeit auffindbaren Stelle im Speicher hinterlegt sein. Diese Stelle heißt „Interrupt-Vektor“.

Da ein Interrupt jederzeit erfolgen kann, hat das Hauptprogramm keine Chance, vor dem Aufruf die Registerinhalte zu sichern. Für Interrupt-Unterprogramme, sog. Interrupt-Handler, ist es daher zwingend notwendig, sämtliche Register vor Verwendung zu sichern und hinterher zurückzuholen.

Beispiel für die Verwendung eines Interrupts: Roboter-Steuerung

Datei: RP6Base/RP6Base\_Examples/RP6Examples\_20080915/RP6Lib/RP6base/RP6RobotBaseLib.c

Suchbegriff: ISR

```
ISR (INT0_vect)
{
    mleft_dist++;
    mleft_counter++;
    /* ... */
}
```

- Durch das Schlüsselwort [ISR](#) anstelle von z. B. [void](#) teilen wir dem Compiler mit, daß es sich um einen Interrupt-Handler handelt, so daß er entsprechenden Code zum Sichern der Registerinhalte einfügt.
- Durch die Namensgebung [INT0\\_vect](#) teilen wir dem Compiler mit, daß er den Interrupt-Vektor Nr. 0 (also den ersten) auf diesen Interrupt-Handler zeigen lassen soll.

(Tatsächlich handelt es sich bei [ISR](#) und [INT0\\_vect](#) um Macros.)

Die Schreibweise ist spezifisch für die Programmierung des Atmel AVR ATmega unter Verwendung der GNU Compiler Collection (GCC). Bei Verwendung anderer Werkzeuge und/oder Prozessoren kann dasselbe Programm völlig anders aussehen. Wie man Interrupt-Handler schreibt und wie man Interrupt-Vektoren setzt, ist ein wichtiger Bestandteil der Dokumentation der Entwicklungswerkzeuge.

Die so geschriebene Funktion wird immer dann aufgerufen, wenn die Hardware den Interrupt Nr. 0 auslöst. Wann das der Fall ist, hängt von der Beschaltung ab. Im Falle des RP6 geschieht es dann, wenn ein Sensor an der linken Raupenkette einen schwarzen Streifen auf der Encoder-Scheibe registriert, also immer dann, wenn sich die linke Raupenkette des Roboters um eine bestimmte Strecke gedreht hat.

Jedesmal wenn sich die Raupenkette um einen Teilstrich weitergedreht hat, werden also zwei Zähler inkrementiert. Wir können dies nutzen, um z. B. durch Auslesen des Zählers [mleft\\_dist](#) die zurückgelegte Entfernung zu messen. (Die RP6-Bibliothek selbst stellt nur eine Zeit- und eine Geschwindigkeitsmessung zur Verfügung.) Wie dies konkret geschehen kann, sei im folgenden vorgestellt.

Methode 1: Verändern des Interrupt-Handlers

- Da die Bibliothek [RP6RobotBase](#) im Quelltext vorhanden ist, können wir sie selbst ändern und in den Interrupt-Handler [ISR \(INT0\\_vect\)](#) einen eigenen Zähler für Sensor-Ticks einbauen.
- Wenn wir diesen zum Zeitpunkt A auf 0 setzen und zum Zeitpunkt B auslesen, erfahren wir, wieviele „Ticks“ der Roboter dazwischen zurückgelegt hat.

Methode 2: Verwenden eines vorhandenen Zählers

- Tatsächlich enthält [ISR \(INT0\\_vect\)](#) bereits zwei Zähler, die bei jedem Sensor-„Tick“ hochgezählt werden: [mleft\\_dist](#) und [mleft\\_counter](#).
- Einer davon ([mleft\\_dist](#)) wird bei jedem [move\(\)](#) auf 0 zurückgesetzt. Für diesen Zähler enthält [RP6RobotBaseLib.h](#) einen „undokumentierten“ Makro [getLeftDistance\(\)](#), um ihn auszulesen.
- Bei sorgfältiger Lektüre von [RP6RobotBaseLib.c](#) erkennt man, daß es unproblematisch ist, den Zähler vom Hauptprogramm aus auf 0 zu setzen. (Dies ist jedoch mit Vorsicht zu genießen: In einer eventuellen Nachfolgeversion der Bibliothek muß dies nicht mehr der Fall sein!)

### Methode 3: Abfrage der Sensoren mittels Busy Waiting

- Alternativ zur Verwendung des Interrupt-Handlers kann man auch von der eigenen Hauptschleife aus den Sensor periodisch abfragen und bei jeder Änderung einen Zähler hochzählen.
- Diese Methode heißt „Busy Waiting“. Sie hat den Vorteil der Einfachheit aber den Nachteil, daß der Prozessor „in Vollzeit“ damit beschäftigt ist, einen Sensor abzufragen, und eventuelle andere Aufgaben nur noch „nebenher“ erledigen kann.
- Wenn aus irgendwelchen Gründen der Interrupt-Mechanismus nicht verwendet werden kann (z.B. weil der Prozessor über kein Interrupt-Konzept verfügt), könnten wir die Lichtschranke alternativ auch mit einem Input-Port verdrahten und mittels Busy Waiting abfragen.  
Dies funktioniert nur dann, wenn die Schleife wirklich regelmäßig den Sensor abfragt. Sobald der Prozessor längere Zeit mit anderen Dingen beschäftigt ist, können beim Busy Waiting Signale der Lichtschranke verlorengehen. Dieses Problem besteht nicht bei Verwendung von Interrupts.

## 4.5 volatile-Variable

Im C-Quelltext fällt auf, daß die Zähler-Variablen `mleft_dist` und `mleft_counter` als `volatile uint16_t mleft_counter` bzw. `volatile uint16_t mleft_dist` deklariert sind anstatt einfach nur als `uint16_t mleft_counter` und `uint16_t mleft_dist`.

Das Schlüsselwort `volatile` teilt dem C-Compiler mit, daß eine Variable immer im Speicher (RAM) aufbewahrt werden muß und nicht in einem Prozessorregister zwischengespeichert werden darf.

Dies ist deswegen wichtig, weil jederzeit ein Interrupt erfolgen kann, der den Wert der Variablen im Speicher verändert. Wenn im Hauptprogramm alle „überflüssigen“ Speicherzugriffe wegoptimiert wurden, erfährt es nichts von der Änderung.

Entsprechendes gilt für I/O-Ports: Wenn ein Programm einen Wert in einen Output-Port schreiben oder aus einem Input-Port lesen soll, ist es wichtig, daß der Speicherzugriff auch tatsächlich stattfindet.

## 4.6 Byte-Reihenfolge – Endianness

### 4.6.1 Konzept

Beim Speichern von Werten, die größer sind als die kleinste adressierbare Einheit (= Speicherzelle oder Speicherwort, häufig 1 Byte), werden mehrere Speicherworte belegt.

Beispiel: 16-Bit-Zahl in 2 8-Bit-Speicherzellen

$$1027 = 1024 + 2 + 1 = 0000\ 0100\ 0000\ 0011_2 = 0403_{16}$$

Diese 16-Bit-Zahl kann auf zwei verschiedene Weisen in zwei 8-Bit-Speicherzellen gespeichert werden:

04	03	<i>Big-Endian</i> , „großes Ende zuerst“, für Menschen leichter lesbar
03	04	<i>Little-Endian</i> , „kleines Ende zuerst“, bei Additionen effizienter (Schriftliches Addieren beginnt immer beim Einer.)

Welche Konvention man verwendet, ist letztlich Geschmackssache und hängt von der verwendeten Hardware (Prozessor) und Software ab. Man spricht hier von der *Endianness* (Byte-Reihenfolge) der Hardware bzw. der Software.

Im Kontext des Datenaustausches ist es wichtig, sich auf eine einheitliche Endianness zu verständigen. Dies gilt insbesondere für:

- Dateiformate
- Datenübertragung

#### 4.6.2 Dateiformate

Als Beispiel für Dateiformate, in denen die Reihenfolge der Bytes in 16- und 32-Bit-Zahlen spezifiziert ist, seien hier Audio-Formate genannt:

- RIFF-WAVE-Dateien (.wav): Little-Endian
- Au-Dateien (.au): Big-Endian
- ältere AIFF-Dateien (.aiff): Big-Endian
- neuere AIFF-Dateien (.aiff): Little-Endian

Insbesondere ist es bei AIFF-Dateien wichtig, zu prüfen, um welche Variante es sich handelt, anstatt sich auf eine bestimmte Byte-Reihenfolge zu verlassen.

Bei Dateiformaten mit variabler Endianness ist es sinnvoll und üblich, die Endianness durch eine Kennung anzuzeigen. Dies geschieht häufig am Anfang der Datei (im „Vorspann“ – „Header“).

Als weiteres Beispiel seien zwei Monochrom-Grafik-Formate genannt. Hier steht jedes Bit für einen schwarzen bzw. weißen Bildpunkt, daher spielt die Reihenfolge der Bits in den Bytes eine entscheidende Rolle. (Diese Grafik-Formate werden in Übungsaufgaben weiter vertieft.)

- PBM-Dateien: Big-Endian, *MSB first* – die höchstwertige Binärziffer ist im Bild links
- XBM-Dateien: Little-Endian, *LSB first* – die höchstwertige Binärziffer ist im Bild rechts

MSB/LSB = most/least significant bit

Achtung: Die Abkürzungen „MSB/LSB“ werden manchmal auch für „most/least significant *byte*“ verwendet. Im konkreten Fall ist es ratsam, die verwendete Byte- und Bit-Reihenfolge genau zu recherchieren bzw. präzise zu dokumentieren.

#### 4.6.3 Datenübertragung

Bei der Übertragung von Daten über Leitungen spielt sowohl die Reihenfolge der Bits in den Bytes („MSB first“ bzw. „LSB first“) als auch die Reihenfolge der Bytes in den übertragenen Zahlen („Big-/Little-Endian“) eine Rolle.

Als Beispiele seien genannt:

- RS-232 (serielle Schnittstelle): MSB first
- I<sup>2</sup>C: LSB first
- USB: beides  
Um Übertragungsfehler erkennen zu können, werden im USB-Protokoll bestimmte Werte einmal gemäß der MSB-first- und einmal gemäß der LSB-first-Konvention übertragen und anschließend auf Gleichheit geprüft.
- Ethernet: LSB first
- TCP/IP (Internet): Big-Endian

Insbesondere gilt für die Übertragung z. B. einer 32-Bit-Zahl über das Internet, daß die vier Bytes von links nach rechts (Big-Endian) übertragen werden, die acht Bits innerhalb jedes Bytes hingegen von rechts nach links (LSB first).

## 4.7 Binärdarstellung von Zahlen

Es gibt unendlich viele verschiedene ganze Zahlen. Da Speicherplatz begrenzt ist, können Rechner nur begrenzt viele Zahlen darstellen. In der Praxis legt man sich auf eine Anzahl von Bits fest, die eine Zahl maximal belegen darf. Typischerweise handelt es sich bei dieser Anzahl von Bits um eine Zweierpotenz: 8, 16, 32, 64.

Eine 8-Bit-Zahl kann per definitionem Binärzahlen von 0000 0000 bis 1111 1111 darstellen. Dezimal sind dies die Zahlen von 0 bis 255. Wenn man die 8-Bit-Zahl 255 inkrementiert (= um 1 erhöht), gibt es einen Überlauf, und das Rechenergebnis ist 0: Die Binärzahl  $1111\ 1111 + 1 = 1\ 0000\ 0000$  hat 9 Bits. Wenn man das oberste Bit abschneidet, bleibt der Wert 0 übrig.

Auf Computern macht man sich dieses Verhalten für die Binärdarstellung negativer Zahlen zunutze: Wenn  $255 + 1$  (dezimal geschrieben, mit 8-Bit-Zahlen berechnet) den Wert 0 ergibt, dann ist 255 dasselbe wie  $-1$ .

Allgemein gilt: Diejenige Zahl  $y$ , die ich auf eine Zahl  $x$  addieren muß, um auf einem  $n$ -Bit-Rechner einen Überlauf und das Rechenergebnis 0 zu erhalten, ist die  *$n$ -Bit-Binärdarstellung von  $-x$* .

Um diese Zahl direkt auszurechnen, geht man folgendermaßen vor:

1. Man invertiert alle Bits der Zahl.  
Aus 0010 1100 (= Binärdarstellung von 44) wird so zum Beispiel 1101 0011.
2. Man addiert 1 zu der erhaltenen Zahl:  
 $1101\ 0011 + 1 = 1101\ 0100$  (= 8-Bit-Binärdarstellung von  $-44$ ).

Diese Darstellung negativer Zahlen heißt *Zweierkomplement*.

Theoretisch kann man jede Zahl bei gegebener Rechengenauigkeit sowohl als positive als auch als negative Zahl interpretieren. Die folgende Konvention hat sich als sinnvoll herausgestellt:

- Entweder betrachtet man alle Zahlen als positiv,
- oder man betrachtet Zahlen, deren oberstes Bit gesetzt ist, als negativ und die anderen als positiv.

Für normale Anwendungen ist die genaue Anzahl der Bits einer Ganzzahl-Variablen unerheblich, sofern der Wertebereich groß genug für die durchzuführenden Rechnungen ist. In diesen Fällen verwendet man in C den Datentyp **int** für vorzeichenbehaftete ganze Zahlen und **unsigned int** (oder kurz: **unsigned**) für vorzeichenlose. Für die Ausgabe mit **printf()** verwendet man **%d** oder **%i** für vorzeichenbehaftete und **%u** für vorzeichenlose ganze Zahlen.

Für spezielle Situationen, in denen die genaue Anzahl der Bits eine Rolle spielt, stellt C (unter Verwendung von **#include <stdint.h>**) spezielle Datentypen bereit:

C-Datentyp	Bits	Vorzeichen	Wertebereich
<code>int8_t</code>	8	ja	$-128, \dots, 127$
<code>uint8_t</code>	8	nein	$0, \dots, 255$
<code>int16_t</code>	16	ja	$-32\,768, \dots, 32\,767$
<code>uint16_t</code>	16	nein	$0, \dots, 65\,535$
<code>int32_t</code>	32	ja	$-2\,147\,483\,648, \dots, 2\,147\,483\,647$
<code>uint32_t</code>	32	nein	$0, \dots, 4\,294\,967\,295$
<code>int64_t</code>	64	ja	$9\,223\,372\,036\,854\,775\,808, \dots, 9\,223\,372\,036\,854\,775\,807$
<code>uint64_t</code>	64	nein	$0, \dots, 18\,446\,744\,073\,709\,551\,615$

Man beachte, daß es keine „allein richtige“ Binärdarstellung einer negativen Zahl gibt; diese hängt vielmehr von der Genauigkeit  $n$  des  $n$ -Bit-Rechenwerks ab. Auf einem 8-Bit-Rechner ist 255 dasselbe wie  $-1$ , auf einem 16-Bit-Rechner ist 255 eine „völlig normale“ Zahl; stattdessen ist 65535 dasselbe wie  $-1$ .

Ebensowenig gibt es einen „allein richtigen“ Zahlenwert eines Bitmusters. Dieser Zahlenwert hängt von der Genauigkeit  $n$  des  $n$ -Bit-Rechenwerks ab und davon, ob man überhaupt negative Zahlen zuläßt oder vielleicht alle Zahlen als positive Zahlen interpretiert.

Beispiel: Für welche Zahl steht der Speicherinhalt 1001 0000 1100 0011 (binär) = 90a3 (hexadezimal)?

Die richtige Antwort auf diese Frage hängt vom Datentyp ab, also von der Bitgenauigkeit des Rechenwerks und davon, ob wir überhaupt mit Vorzeichen rechnen:

```
als int8_t:           -93   (nur unteres Byte, Little-Endian)
als uint8_t:          163   (nur unteres Byte, Little-Endian)
als int16_t:         -28509
als uint16_t:         37027
int32_t oder größer: 37027  (zusätzliche Bytes mit Nullen aufgefüllt)
```

Siehe auch: <http://xkcd.com/571/>

## 4.8 Speicherausrichtung – Alignment

Ein 32-Bit-Prozessor kann auf eine 32-Bit-Variable effizienter zugreifen, wenn die Speicheradresse der Variablen ein Vielfaches von 32 Bits, also 4 Bytes ist. Eine Variable, auf die dies zutrifft, heißt „korrekt im Speicher ausgerichtet“ („aligned“).

„Effizienter“ kann bedeuten, daß Maschinenbefehle zum Arbeiten mit den Variablen schneller abgearbeitet werden. Es kann aber auch bedeuten, daß der Prozessor gar keine direkte Bearbeitung von inkorrekt ausgerichteten Variablen erlaubt. In diesem Fall bedeutet eine inkorrekte Speicherausrichtung, daß für jede Operation mit der Variablen anstelle eines einzelnen Maschinenbefehls ein kleines Programm aufgerufen werden muß.

Um zu verstehen, welche Konsequenzen dies für die Arbeit mit Rechnern hat, betrachten wir die folgenden Variablen:

```
#include <stdint.h>
```

```
uint8_t a;
uint16_t b;
uint8_t c;
```

Die Anordnung dieser Variablen im Speicher könnte z. B. folgendermaßen aussehen:

	...
3005	
3004	
3003	c
3002	b
3001	
3000	a
2fff	
	...

Ein optimierender Compiler wird für eine korrekte Ausrichtung der Variablen **b** sorgen, beispielsweise durch Auffüllen mit unbenutzten Speicherzellen:

	...
3005	
3004	c
3003	
3002	b
3001	
3000	a
2fff	
	...

Alternativ ist es dem Compiler auch möglich, die korrekte Ausrichtung durch „Umsortieren“ der Variablen herzustellen und dadurch „Löcher“ zu vermeiden:

	...
3005	
3004	
3003	
3002	b
3001	c
3000	a
2fff	
	...

Fazit: Man kann sich als Programmierer nicht immer darauf verlassen, daß die Variablen im Speicher in einer spezifischen Weise angeordnet sind.

In vielen existierenden Programmen geschieht dies dennoch. Diese Programme sind fehlerhaft. Dort kann es z. B. passieren, daß nach einem Upgrade des Compilers schwer lokalisierbare Fehler auftreten.

Entsprechende Überlegungen gelten für 64-Bit- und 16-Bit-Prozessoren. Die Größe der Variablen, aufgerundet auf die nächste Zweierpotenz, gibt eine Ausrichtung vor. Die Registerbreite des Prozessors markiert die größte Ausrichtung, die noch berücksichtigt werden muß.

Bei 8-Bit-Prozessoren stellt sich die Frage nach der Speicherausrichtung normalerweise nicht, weil die kleinste adressierbare Einheit eines Speichers selten kleiner als 8 Bits ist.

Beispiele:

- Eine 64-Bit-Variable auf einem 64-Bit-Prozessor muß auf 64 Bits ausgerichtet sein.
- Eine 32-Bit-Variable auf einem 64-Bit-Prozessor braucht nur auf 32 Bits ausgerichtet zu sein.
- Eine 64-Bit-Variable auf einem 32-Bit-Prozessor braucht nur auf 32 Bits ausgerichtet zu sein.
- Eine 64-Bit-Variable auf einem 8-Bit-Prozessor braucht nur auf 8 Bits ausgerichtet zu sein.

Bei der Definition von Datenformaten tut man gut daran, die Ausrichtung der Daten von vorneherein zu berücksichtigen, um auf möglichst vielen – auch zukünftigen – Prozessoren eine möglichst effiziente Bearbeitung zu ermöglichen.

Wenn ich beispielsweise ein Dateiformat definiere, in dem 128-Bit-Werte vorkommen, ist es sinnvoll, diese innerhalb der Datei auf 128 Bits (16 Bytes) auszurichten, auch wenn mein eigener Rechner nur über einen 64-Bit-Prozessor verfügt.

## 5 Algorithmen

### 5.1 Differentialgleichungen

Eine mathematische Gleichung mit einer gesuchten Zahl  $x$ , z. B.

$$x + 2 = -x$$

läßt sich leicht nach der Unbekannten  $x$  auflösen. (In diesem Fall lautet die Lösung:  $x = -1$ .)

Wesentlich schwieriger ist es, eine mathematische Gleichung mit einer gesuchten Funktion  $x(t)$  zu lösen, z. B.:

$$x'(t) = -x(t) \quad \text{mit} \quad x(0) = 1$$

Um hier auf die Lösung  $x(t) = e^{-t}$  zu kommen, sind bereits weitreichende mathematische Kenntnisse erforderlich.

Eine derartige Gleichung, die einen Zusammenhang zwischen der gesuchten Funktion und ihren Ableitungen vorgibt, heißt *Differentialgleichung*. Viele physikalisch-technische Probleme werden durch Differentialgleichungen beschrieben.

### 5.1.1 Beispiel: Pendelschwingung

Das Verhalten eines Fadenpendels (mathematisches Pendel) wird durch seine Auslenkung  $\phi$  als Funktion der Zeit  $t$  beschrieben. Wie kann man  $\phi(t)$  berechnen?

Wie aus anderen Veranstaltungen (Grundlagen der Physik, Mechanik) her bekannt sein sollte, wirkt auf ein Fadenpendel, das um den Winkel  $\phi(t)$  ausgelenkt ist, die tangentielle Kraft  $F = -m \cdot g \cdot \sin \phi(t)$ . Gemäß der Formel  $F = m \cdot a$  bewirkt diese Kraft eine tangentielle Beschleunigung  $a = -g \cdot \sin \phi(t)$ . (Das Minuszeichen kommt daher, daß die Kraft der Auslenkung entgegengesetzt wirkt.)

Wenn das Pendel die Länge  $l$  hat, können wir dieselbe tangentielle Beschleunigung mit Hilfe der zweiten Ableitung des Auslenkungswinkels  $\phi(t)$  berechnen:  $a = l \cdot \phi''(t)$  (Winkel in Bogenmaß). Durch Gleichsetzen erhalten wir eine Gleichung, die nur noch eine Unbekannte enthält, nämlich die Funktion  $\phi(t)$ .

Um  $\phi(t)$  zu berechnen, müssen wir also die Differentialgleichung

$$\phi''(t) = -\frac{g}{l} \cdot \sin \phi(t)$$

lösen.

Diese Differentialgleichung läßt sich mit „normalen“ Mitteln nicht lösen, daher verwendet man in der Praxis meistens die Kleinwinkelnäherung  $\sin \phi \approx \phi$  (für  $\phi \ll 1$ ) und löst stattdessen die Differentialgleichung:

$$\phi''(t) = -\frac{g}{l} \cdot \phi(t)$$

Für ein mit der Anfangsauslenkung  $\phi(0)$  losgelassenes Pendel lautet dann das Ergebnis:

$$\phi(t) = \phi(0) \cdot \cos(\omega t) \quad \text{mit} \quad \omega = \sqrt{\frac{g}{l}}$$

Das Beispielprogramm [pendulum-1.c](#) illustriert, welche Bewegung sich aus diesem  $\phi(t)$  ergibt.

### 5.1.2 Das explizite Euler-Verfahren

Um eine Differentialgleichung mit Hilfe eines Computers näherungsweise *numerisch* zu lösen, stehen zahlreiche Lösungsverfahren zur Verfügung. Im folgenden soll das einfachste dieser Verfahren, das *explizite Euler-Verfahren* (auch *Eulersches Polygonzugverfahren* genannt) vorgestellt werden.

Wir betrachten das System während eines kleinen Zeitintervalls  $\Delta t$ . Während dieses Zeitintervalls sind alle von der Zeit  $t$  abhängigen Funktionen – z. B. Ort, Geschwindigkeit, Beschleunigung, Kraft – näherungsweise konstant.

Bei konstanter Geschwindigkeit  $v$  ist es einfach, aus dem Ort  $x(t)$  zu Beginn des Zeitintervalls den Ort  $x(t + \Delta t)$  am Ende des Zeitintervalls zu berechnen:

$$x(t + \Delta t) = x(t) + \Delta t \cdot v$$

Bei konstanter Kraft  $F = m \cdot a$  und somit konstanter Beschleunigung  $a$  ist es ebenso einfach, aus der Geschwindigkeit  $v(t)$  zu Beginn des Zeitintervalls die Geschwindigkeit  $v(t + \Delta t)$  am Ende des Zeitintervalls zu berechnen:

$$v(t + \Delta t) = v(t) + \Delta t \cdot a$$

Wenn wir dies in einer Schleife durchführen und jedesmal  $t$  um  $\Delta t$  erhöhen, erhalten wir Näherungen für die Funktionen  $x(t)$  und  $v(t)$ .

Für das oben betrachtete Beispiel (Fadenpendel) müssen wir in jedem Zeitintervall  $\Delta t$  zunächst die tangentielle Beschleunigung  $a(t)$  aus der tangentialen Kraft berechnen, die sich wiederum aus der momentanen Auslenkung  $\phi(t)$  ergibt:

$$a = -g \cdot \sin \phi$$

Mit Hilfe dieser – innerhalb des Zeitintervalls näherungsweise konstanten – Beschleunigung berechnen wir die neue tangentielle Geschwindigkeit:

$$v(t + \Delta t) = v(t) + \Delta t \cdot a$$

Mit Hilfe dieser – innerhalb des Zeitintervalls näherungsweise konstanten – Geschwindigkeit berechnen wir schließlich die neue Winkelauslenkung  $\phi$ , wobei wir einen kleinen Umweg über den Kreisbogen  $x = l \cdot \phi$  machen:

$$\phi(t + \Delta t) = \frac{x(t + \Delta t)}{l} = \frac{x(t) + \Delta t \cdot v}{l} = \phi(t) + \frac{\Delta t \cdot v}{l}$$

Ein C-Programm, das diese Berechnungen durchführt (Datei: [pendulum-2.c](#)), enthält als Kernstück:

```
#define g 9.81
#define l 1.0
#define dt 0.05
#define phi0 30.0 /* degrees */

float t = 0.0;
float phi = phi0 * M_PI / 180.0;
float v = 0.0;

void calc (void)
{
    float a = -g * sin (phi);
    v += dt * a;
    phi += dt * v / l;
}
```

Jeder Aufruf der Funktion `calc()` versetzt das Pendel um das Zeitintervall `dt` in die Zukunft.

Es ist vom Verfahren her nicht notwendig, mit der Kleinwinkelnäherung  $\sin \phi \approx \phi$  zu arbeiten. Das Beispielprogramm [pendulum-3.c](#) illustriert, welchen Unterschied die Kleinwinkelnäherung ausmacht.

Wie gut arbeitet das explizite Euler-Verfahren? Um dies zu untersuchen, lösen wir eine Differentialgleichung, deren exakte Lösung aus der Literatur bekannt ist, nämlich die Differentialgleichung mit Kleinwinkelnäherung. Das Beispielprogramm [pendulum-4.c](#) vergleicht beide Lösungen miteinander. Für das betrachtete Beispiel ist die Übereinstimmung recht gut; für Präzisionsberechnungen ist das explizite Euler-Verfahren jedoch nicht genau (und stabil) genug. Hierfür sei auf die Lehrveranstaltungen zur numerischen Mathematik verwiesen.

## Bemerkung

Das Beispielprogramm [pendulum-4.c](#) berechnet mit überzeugender Übereinstimmung dasselbe Ergebnis für die Auslenkung des Pendels auf zwei verschiedene Weisen:

1. über eine Formel, die einen Cosinus enthält,
2. mit Hilfe der Funktion `calc()`, die nur Grundrechenarten verwendet.

Dies läßt die Natur der Verfahren erahnen, mit deren Hilfe es möglich ist, Sinus, Cosinus und andere kompliziertere Funktionen nur unter Verwendung der Grundrechenarten zu berechnen.



## 5.2 Rekursion

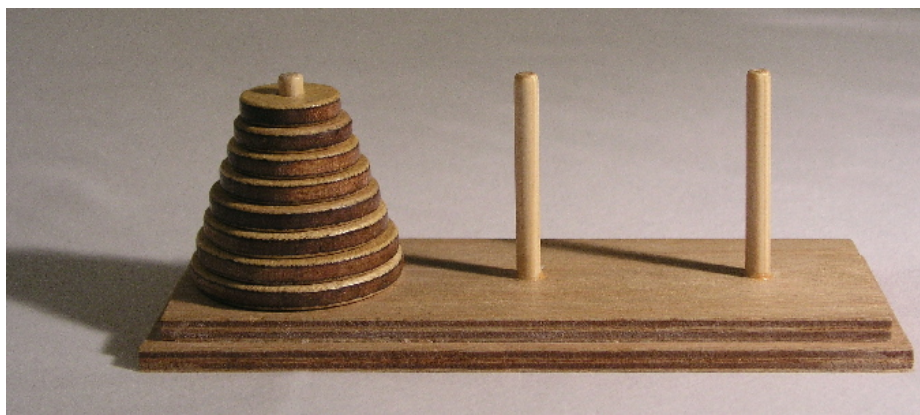
Aus der Mathematik ist das Beweisprinzip der *vollständigen Induktion* bekannt:

$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$$

Wenn auf diese Weise die Lösbarkeit eines Problems bewiesen wurde, ist es direkt möglich, das Problem im Computer *tatsächlich* zu lösen, nämlich durch einen *rekursiven Algorithmus*.

Ein klassisches Beispiel für ein rekursiv lösbares Problem sind die Türme von Hanoi:

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.



Bildquelle: [http://commons.wikimedia.org/wiki/File:Tower\\_of\\_Hanoi.jpeg](http://commons.wikimedia.org/wiki/File:Tower_of_Hanoi.jpeg)  
Urheber: <http://en.wikipedia.org/wiki/User:Evanherk>  
Lizenz: GNU FDL (Version 1.2 oder später) oder  
Creative Commons Attribution-Share Alike (Version 3.0 Unported)

Die rekursive Lösung des Problems lautet:

- Wenn  $n = 1$  ist, also nur eine Scheibe vorliegt, läßt sich diese „einfach so“ an den Zielplatz verschieben. In diesem Fall sind wir direkt fertig.
- Wenn  $n - 1$  Scheiben als verschiebbar vorausgesetzt werden, lautet die Vorgehensweise:  
verschiebe die oberen  $n - 1$  Scheiben auf einen Hilfsplatz,  
verschiebe die darunterliegende einzelne Scheibe auf den Zielplatz,  
verschiebe die  $n - 1$  Scheiben vom Hilfsplatz auf den Zielplatz.

Dieser Algorithmus läßt sich unmittelbar in eine Programmiersprache übersetzen:

```
void verschiebe (int n, int start, int ziel)
{
    if (n == 1)
        verschiebe_1_scheibe (start, ziel);
    else
    {
        verschiebe (1, start, hilfsplatz);
        verschiebe (n - 1, start, ziel);
        verschiebe (1, hilfsplatz, ziel);
    }
}
```

## 5.3 Aufwandsabschätzungen

### 5.3.1 Sortieralgorithmen

Am Beispiel von Sortieralgorithmen soll hier aufgezeigt werden, wie man die Lösung eines Problems schrittweise effizienter gestalten kann.

Als Problem wählen wir das Sortieren eines Arrays (z. B. von Namen).

- Minimum/Maximum ermitteln: [sort-0.c](#), [sort-1.c](#) und [sort-2.c](#)
- Selectionsort: [sort-3.c](#), [sort-4.c](#)
- Bubblesort: [sort-5.c](#)
- Bubblesort mit Abbruch in äußerer Schleife: [sort-6.c](#)
- Bubblesort mit Abbruch in innerer Schleife: [sort-7.c](#), [sort-7a.c](#), [sort-7b.c](#)
- Vergleich mit Quicksort: [sort-8.c](#), [sort-8a.c](#), [sort-8b.c](#)

Bei „zufällig“ sortierten Ausgangsdaten arbeitet Quicksort schneller als Bubblesort. Wenn die Ausgangsdaten bereits nahezu sortiert sind, ist es umgekehrt. Im jeweils ungünstigsten Fall arbeiten beide Algorithmen gleich langsam.

### 5.3.2 Landau-Symbole

Das Landau-Symbol  $\mathcal{O}(g)$  mit einer Funktion  $g(n)$  steht für die *Ordnung* eines Algorithmus', also die „Geschwindigkeit“, mit der er arbeitet. Die Variable  $n$  bezeichnet die Menge der Eingabedaten, hier also z. B. die Anzahl der Namen.

- $\mathcal{O}(n)$  bedeutet, daß die Rechenzeit mit der Menge der Eingabedaten linear wächst. Um doppelt so viele Namen zu sortieren, benötigt das Programm doppelt so lange.
- $\mathcal{O}(n^2)$  bedeutet, daß die Rechenzeit mit der Menge der Eingabedaten quadratisch wächst. Um doppelt so viele Namen zu sortieren, benötigt das Programm viermal so lange.
- $\mathcal{O}(2^n)$  bedeutet, daß die Rechenzeit mit der Menge der Eingabedaten exponentiell wächst. Für jeden Namen, der dazukommt, benötigt das Programm doppelt so lange.  
Ein derartiges Programm gilt normalerweise als inakzeptabel langsam.
- $\mathcal{O}(\log n)$  bedeutet, daß die Rechenzeit mit der Menge der Eingabedaten logarithmisch wächst. Für jede Verdopplung der Namen benötigt das Programm nur einen Rechenschritt mehr.  
Ein derartiges Programm gilt als „traumhaft schnell“. Dies wird jedoch nur selten erreicht.
- $\mathcal{O}(1)$  bedeutet, daß die Rechenzeit von der Menge der Eingabedaten unabhängig ist: 1 000 000 Namen werden genau so schnell sortiert wie 10.  
Dies ist nur in Ausnahmefällen erreichbar.
- $\mathcal{O}(n \log n)$  liegt zwischen  $\mathcal{O}(n)$  und  $\mathcal{O}(n^2)$ .  
Ein derartiges Programm gilt als schnell. Viele Sortieralgorithmen erreichen dieses Verhalten.

Wie sieht man einem Programm an, wie schnell es arbeitet?

- Vorfaktoren interessieren nicht.  
Wenn ein Code immer – also unabhängig von den Eingabedaten – zweimal ausgeführt wird, beeinflusst das die Ordnung des Algorithmus nicht.  
Wenn ein Code immer – also unabhängig von den Eingabedaten – 1 000 000mal ausgeführt wird, mag das Programm für kleine Datenmengen langsam erscheinen. Für die Ordnung interessiert jedoch nur das Verhalten für große Datenmengen, und dort kann dasselbe Programm durchaus schnell sein.
- Jede Schleife, die von 0 bis  $n$  geht, multipliziert die Rechenzeit des innerhalb der Schleife befindlichen Codes mit  $n$ .  
Eine Doppelschleife (Schleife innerhalb einer Schleife) hat demnach  $\mathcal{O}(n^2)$ .

- Wenn sich die Grenzen einer Schleife ständig ändern, nimmt man den Durchschnitt.

Beispiel:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
        ...
```

Die äußere Schleife wird immer  $n$ -mal ausgeführt, die innere *im Durchschnitt*  $\frac{n}{2}$ -mal, was proportional zu  $n$  ist.

Zusammen ergibt sich  $\mathcal{O}(n^2)$ .

- Bei Rekursionen muß man mitzählen, wie viele Schleifen hinzukommen.

Bei Quicksort wird z. B. in jeder Rekursion eine Schleife von 0 bis  $n$  (aufgeteilt) ausgeführt. Bei jeder Rekursion wird das Array „normalerweise“ halbiert, d. h. die Rekursionstiefe ist proportional zum Logarithmus von  $n$  (zur Basis 2). Daraus ergibt sich die Ordnung  $\mathcal{O}(n \log n)$  für den „Normalfall“ des Quicksort. (Im ungünstigsten Fall kann sich auch  $\mathcal{O}(n^2)$  ergeben.)

Für eine präzise Definition der Landau-Symbole siehe z. B.: <http://de.wikipedia.org/wiki/Landau-Symbole>

## 6 Objektorientierte Programmierung

### 6.0 Dynamische Speicherverwaltung

Variable in C haben grundsätzlich eine feste Größe. Dies gilt auch für Arrays: Auch mit der Schreibweise ohne Größenangabe, z. B. `int a[] = { 2, 3, 5, 7 };` handelt es sich *nicht* um ein Array veränderlicher Größe. Die `[]`-Schreibweise besagt lediglich, daß der Compiler die Größe des Arrays anhand des Initialisierers (hier: `{ 2, 3, 5, 7 }`) selbst berechnen soll. Das Beispiel `int a[] = { 2, 3, 5, 7 };` ist nur eine andere Schreibweise für `int a[4] = { 2, 3, 5, 7 };`.

Um *tatsächlich* Arrays mit einer variablen Anzahl von Elementen verwenden zu können, ist es in C notwendig, durch einen Funktionsaufruf explizit Speicher zu reservieren:

```
#include <stdlib.h>
...
int *a = malloc (4 * sizeof (int));
...
free (a);
```

`malloc()` reserviert Speicher, `free()` gibt ihn wieder frei.

Man beachte, daß man in C auf Zeiger mit dem `[]`-Operator genau wie auf „normale“ Arrays mit einem Index zugreifen kann:

```
int *a = malloc (4 * sizeof (int));
...
for (int i = 0; i < 4; i++)
    printf ("%d\n", a[i]);
```

Es gibt normalerweise keine Möglichkeit, einem Zeiger (hier: `a`) anzusehen, wie groß der Speicherbereich ist, auf den er zeigt. Diesen Wert muß sich das Programm selbst merken, typischerweise in einer Variablen:

```
int n = 4;
int *a = malloc (n * sizeof (int));
...
for (int i = 0; i < n; i++)
    printf ("%d\n", a[i]);
```

## 6.1 Konzepte und Ziele

Für viele Anwendungen ist der o. a. Mechanismus der *dynamischen Arrays* noch nicht flexibel genug: Auch wenn die Anzahl der Elemente nicht mehr festliegt, so müssen doch alle genau dieselbe Größe haben. In vielen Situationen möchte man jedoch eine vorher nicht festgelegte Anzahl von Objekten unterschiedlichen Typs in einer Schleife abarbeiten – z. B. verschiedene geometrische Objekte in einem Zeichenprogramm oder verschiedene Bedienelemente (Button, Zeichenbereich, Eingabefeld, ...) in einer graphischen Benutzeroberfläche (Graphical User Interface – GUI).

Um dieses Problem zu lösen, speichert man Zeiger auf Objekte unterschiedlicher Größe in einem dynamischen Array. Dies funktioniert, weil alle Zeiger – auch wenn sie auf unterschiedlich große Objekte zeigen – die gleiche Größe haben und daher in demselben Array koexistieren können.

Um alle diese Objekte in einer Schleife auf gleiche Weise behandeln zu können, benötigt man standardisierte Funktionen, die mit dem Objekt arbeiten. Diese nennt man *Methoden*.

Eine Methode bewirkt unterschiedliche Dinge, je nachdem, auf welches Objekt sie angewandt wird. Dies hängt vom Typ des Objekts ab. Um dies zu realisieren, kann man ein Objekt als **struct**-Variable speichern, die zusätzlich zum eigentlichen Inhalt eine Kennung für den Objekttyp als Datenfeld enthält. In der Methode fragt man diese Typkennung ab und entscheidet auf dieser Grundlage, was die Methode bewirkt.

Dies kann über **if**-Abfragen (oder **switch**-Anweisungen) geschehen, bei sehr vielen unterschiedlichen Objekttypen entarten derartige Methoden jedoch zu sehr unübersichtlichen **if**-Ketten. Weiter unten werden elegantere Wege zur Realisierung von Methoden vorgestellt.

Objekte, die einen gemeinsamen Anteil von Eigenschaften haben und sich typischerweise in demselben Array befinden, bezeichnet man als *miteinander verwandt*. Der „kleinste gemeinsame Nenner“ dieser Objekte, also ein Objekttyp der *nur* den gemeinsamen Anteil enthält, heißt *Basisklasse* oder *gemeinsamer Vorfahr* der Objekte. Umgekehrt heißt ein Objekttyp, der eine Basisklasse um neue Eigenschaften erweitert, *abgeleitete Klasse* oder *Nachfahre* der Basisklasse.

Eigenschaften, die ein Objekttyp mit seinem Vorfahren gemeinsam hat, bezeichnet man als *vom Vorfahren geerbt*.

Ein „Array von Objekten“ wird zunächst als Array von Zeigern auf die Basisklasse realisiert; die Zeiger zeigen aber in Wirklichkeit auf Objekte von abgeleiteten Klassen. Diese Möglichkeit, unterschiedliche Objekte gemeinsam zu verwalten, bezeichnet man als *Polymorphie* (griechisch: *Vielgestaltigkeit*).

## 6.2 Beispiel: Zahlen und Buchstaben

Als Beispiel konstruieren wir eine Struktur, die Zahlen und Buchstaben (Strings) gemeinsam verwalten soll, also gewissermaßen ein Array, das sowohl ganze Zahlen (**int**) als auch Strings (**char \***) als Elemente haben kann.

Zu diesem Zweck definieren wir zwei **struct**-Datentypen **t\_integer** und **t\_string**, die als Inhalt (**content**) eine ganze Zahl bzw. einen String enthalten und zusätzlich eine Typ-Kennung (hier: **int type**). Weiterhin definieren wir einen gemeinsamen Vorfahren **t\_base**, der *nur* die Typ-Kennung enthält.

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

Man beachte, daß diese drei **struct**-Datentypen trotz der absichtlichen Gemeinsamkeiten aus Sicht des C-Compilers nichts miteinander zu tun haben; sie sind voneinander vollkommen unabhängige Datentypen.

Unser „Array von Zahlen und Buchstaben“ erzeugen wir nun als Array von Zeigern auf den Basistyp, lassen die Zeiger aber in Wirklichkeit auf Variablen der abgeleiteten Datentypen zeigen:

```
#define T_INTEGER 1
#define T_STRING 2

t_integer i = { T_INTEGER, 42 };
t_string s = { T_STRING, "Hello, world!" };

t_base *object[] = { (t_base *) &i, (t_base *) &s, NULL };
                    explizite
                    Typumwandlung
```

Damit der Compiler dies ohne Warnung akzeptiert, ist eine explizite Typumwandlung des jeweiligen Zeigers auf den abgeleiteten Typ in einen Zeiger auf den Basistyp erforderlich.

Bei der Benutzung der abgeleiteten Typen erfolgt wieder eine explizite Typumwandlung, nur diesmal in umgekehrter Richtung:

```
void print_object (t_base *this)
{
    if (this->type == T_INTEGER)
        printf ("Integer: %d\n", ((t_integer *) this)->content);
    else if (this->type == T_STRING)
        printf ("String: %s\n", ((t_string *) this)->content);
}
...
for (int i = 0; object[i]; i++)
    print_object (object[i]);
```

(Beispiel-Programm: [objects-7.c](#))

Die expliziten Typumwandlungen sind ein gravierender Nachteil dieser Vorgehensweise, denn sie schalten jegliche Überprüfung durch den Compiler aus. Der Programmierer ist komplett selbst dafür verantwortlich, daß die **struct**-Datentypen gemeinsame Felder haben und daß der Zeiger jeweils auf den richtigen **struct**-Typ zeigt.

Die folgenden Abschnitte stellen Möglichkeiten vor, diese Nachteile abzumildern.

Die Verwendung von Zeigern auf „normale“ Variable ist in der Praxis unüblich. Stattdessen reserviert man mit `malloc()` Speicher für die Objekte. Es hat sich bewährt, für diesen Zweck eine spezielle Funktion, den sog. *Konstruktor* zu schreiben. Der Konstruktor kann den reservierten Speicher auch direkt mit sinnvollen Werten initialisieren, wodurch wieder eine Fehlerquelle wegfällt.

```
t_integer *new_integer (int i)
{
    t_integer *p = malloc (sizeof (t_integer));
    p->type = T_INTEGER;
    p->content = i;
    return p;
}

t_string *new_string (char *s)
{
    t_string *p = malloc (sizeof (t_string));
    p->type = T_STRING;
```

```

    p->content = s;
    return p;
}
...

t_base *object[] = { (t_base *) new_integer (42),
                     (t_base *) new_string ("Hello,_world!"),
                     NULL };

```

(Beispiel-Programm: [objects-8.c](#))

## 6.3 Unions

Explizite Typumwandlungen sind unsicher und nach Möglichkeit zu vermeiden. Eine Alternative ergibt sich durch Verwendung des Datentyps **union**.

Eine **union** sieht formal wie ein **struct** aus. Der Unterschied besteht darin, daß die Datenfelder eines **struct** im Speicher *hintereinander* liegen, wohingegen sich die Datenfelder einer **union** *denselben Speicherbereich teilen*.

```

#include <stdio.h>
#include <stdint.h>

typedef union
{
    int8_t i;
    uint8_t u;
} num8_t;

int main (void)
{
    num8_t n;
    n.i = -3;
    printf ("%d\n", n.u);
    return 0;
}

```

Die im o. a. Beispiel konstruierte **union** spricht dieselbe Speicherzelle einerseits als **int8\_t** an und andererseits als **uint8\_t**. Das Beispiel-Programm (Datei: [unions-1.c](#)) nutzt dies aus, um die negative Zahl **-3** als positive 8-Bit-Zahl auszugeben (Berechnung des Zweierkomplements).

In der objektorientierten Programmierung in C nutzt man **union**-Datentypen, um ohne explizite Typumwandlung denselben Speicherbereich als Objekte verschiedenen Typs zu interpretieren:

```

typedef struct
{
    int type;
} t_base;

typedef struct
{
    int type;
    int content;
} t_integer;

typedef struct
{
    int type;
    char *content;
} t_string;

typedef union
{
    t_base base;
    t_integer integer;
    t_string string;
} t_object;

if (this->base.type == T_INTEGER)
    printf ("Integer:_%d\n", this->integer.content);
else if (this->base.type == T_STRING)
    printf ("String:_%s\n", this->string.content);

```

(Beispiel-Programm: [objects-9.c](#))

Das Ansprechen falscher Speicherbereiche wird hierdurch zwar nicht völlig ausgeschlossen; der Compiler hat jedoch wesentlich mehr Möglichkeiten als bei expliziten Typumwandlungen, den Programmierer vor derartigen Fehlern zu bewahren.

Das Problem, von Hand dafür sorgen zu müssen, daß die **struct**-Datentypen zueinander passende Datenfelder enthalten, bleibt weiterhin bestehen.

Ein alternativer Ansatz besteht darin, nur die veränderlichen Eigenschaften der Objekte in einer **union** zu speichern – siehe Aufgabe 1 (c) bis (e) in den Übungen vom 19. 12. 2016 (Datei: [hp-uebung-20161219.pdf](#)).

## 6.4 Beispiel: graphische Benutzeroberfläche (GUI)

**GTK+** ist eine weit verbreitete Bibliothek zur Erstellung graphischer Benutzeroberflächen (Graphical User Interface – GUI). Sie wurde ursprünglich für das Bildverarbeitungsprogramm **GIMP** geschrieben, kommt aber u. a. auch im Web-Browser **Mozilla Firefox** zum Einsatz.

GTK+ ist in C geschrieben und objektorientiert. Die Bibliothek verwendet intern einige der hier besprochenen Vorgehensweisen zur Realisierung objektorientierter Programmierung in C.

Die Beispielprogramme [gtk-1.c](#) bis [gtk-7.c](#) demonstrieren, wie man mit Hilfe von GTK+ ein einfaches GUI-Programm schreibt, das graphische Objekte (Rechteck, Kreis, Dreieck) auf den Bildschirm zeichnet und sich nach dem Anklicken eines Buttons beendet.

Die Bedienelemente der GUI sind in GTK+ Objekte. Hier ein paar Beispiele:

- **GtkWidget** ist die Basisklasse für alle GUI-Elemente.
- **GtkContainer** ist ein Nachfahre von **GtkWidget**.  
Dieses Objekt kann, ähnlich einem Array, andere Objekte enthalten.
- **GtkWindow** ist ein Fenster auf dem Bildschirm.  
Es ist ein Nachfahre von **GtkContainer** und kann daher andere Objekte enthalten.
- **GtkDrawingArea** ist ein rechteckiger Zeichenbereich auf dem Bildschirm.
- **GtkButton** ist ein Bedienknopf.  
Durch Anklicken des Knopfes kann der Benutzer Aktionen auslösen.

Um bei einer abgeleiteten Klasse (z. B. **GtkWindow**) Eigenschaften der Basisklasse (z. B. **GtkContainer**) nutzen zu können, verwendet GTK+ explizite Typumwandlungen, die in Präprozessor-Macros gekapselt sind. Um zum Beispiel ein **GtkWindow**-Objekt **window** als **GtkContainer** ansprechen zu können, verwendet man **GTK\_CONTAINER (window)**.

Ähnlich wie in OpenGL/GLUT erfolgt auch in GTK+ das Zeichnen sowie die Verarbeitung von Benutzereingaben (Tastatur, Maus) über Callbacks.

In **Praktikumsversuch 4** haben Sie selbst weitere Erfahrungen mit GTK+ gesammelt und gleichzeitig eine eigene Objekt-Hierarchie (für graphische Objekte: Rechteck, Kreis, Dreieck) programmiert.

## 6.5 Virtuelle Methoden

In großen Programmen wird die Anzahl der verwendeten Objekt-Datentypen schnell sehr groß. Eine Methode, die per **if** unterscheiden muß, welche Art von Objekt sie gerade bearbeitet, enthält dann lange **if**-Ketten und wird dadurch sehr unübersichtlich.

```
void print_object (t_object *this)
{
    if (this->base.type == T_INTEGER)
        printf ("Integer:_%d\n", this->integer.content);
    else if (this->base.type == T_STRING)
        printf ("String:_%s\n", this->string.content);
}
```

Es wäre vorteilhaft, wenn alle Methoden, die sich auf einen bestimmten Objekttyp beziehen, auch nebeneinander im Quelltext stehen könnten, anstatt sich über den gesamten Quelltext zu verteilen (weil jede Funktion einen **if**-Zweig für diesen Objekttyp hat).



```

void print_integer (t_object *this)
{
    printf ("Integer:_%d\n", this->integer.content);
}

void print_string (t_object *this)
{
    printf ("String:_%s\n", this->string.content);
}

```

Um dies zu realisieren, verwendet man *Zeiger auf Funktionen*:

```

typedef struct
{
    void (*print) (union t_object *this);
} t_base;

typedef struct
{
    void (*print) (union t_object *this);
    int content;
} t_integer;

typedef struct
{
    void (*print) (union t_object *this);
    char *content;
} t_string;

```

Um in C einen Zeiger auf eine Funktion zu deklarieren, deklariert man eine „normale“ Funktion, deren „Name“ die Gestalt `(*print)` hat – mit dem vorangestellten Stern und den umschließenden Klammern. (Merkregel: Das, worauf `print` zeigt – also `*print` –, ist eine Funktion.)

Der Aufruf einer derartigen Funktion erfolgt über den im Objekt gespeicherten Zeiger:

```

for (int i = 0; object[i]; i++)
    object[i]->print (object[i]);

```

Eine derartige Funktion, die für verschiedene Objekttypen existiert und bei deren Aufruf automatisch „die passende“ Funktion ausgewählt wird, heißt *virtuelle Methode*.

Jeder Methode wird ein Zeiger `t_object *this` auf das Objekt als Parameter mitgegeben.

Bei der Deklaration der virtuellen Methode innerhalb des Objekt-Typs wird daher der Union-Typ `t_object` bereits benötigt. Dieser kann jedoch erst später deklariert werden, wenn die darin enthaltenen Objekt-Typen bekannt sind.

Um dieses Problem zu lösen, muß die `union t_object` „doppelt“ deklariert werden:

```

typedef union t_object
{
    t_base base;
    t_integer integer;
    t_string string;
} t_object;

```

Dadurch daß `t_object` auch oben, hinter dem Schlüsselwort `union` steht, ist neben dem Datentyp `t_object` auch ein Datentype `union t_object` (in einem separaten Namensraum) bekannt. Derartig deklarierte Typen kann man *vorwärts-deklarierten*: Wenn man eine Zeile

```

union t_object;

```



den anderen Deklarationen voranstellt, wissen diese, daß es später eine `union t_object` geben wird. Zeiger auf diese – noch unbekannte – `union` dürfen dann bereits in Deklarationen – hier: Funktionsparameter – verwendet werden.

Das Beispiel-Programm `objects-12.c` illustriert, wie man virtuelle Methoden in C realisieren kann.

In größeren Projekten ist es nicht effizient, in jeder einzelnen Objektinstanz (= Variable des Objekttyps) sämtliche Zeiger auf sämtliche virtuellen Methoden zu speichern. Stattdessen speichert man in der Objektinstanz lediglich einen Zeiger auf eine Tabelle von Zeigern auf die virtuellen Methoden, die sog. *virtuelle Methodentabelle* – siehe das Beispiel-Programm `objects-13.c`.

## 6.6 Einführung in C++

Objektorientierte Programmierung in C ist sehr mühselig und fehleranfällig: Objekt-Datentypen müssen manuell so abgeglichen werden, daß sie in ihren ersten Datenfeldern übereinstimmen, Konstruktoren müssen manuell erstellt werden, usw.

Um diese Probleme zu beheben, wurden neue Computersprachen entwickelt, die objektorientierte Programmierung durch neue Sprachelemente unterstützen. Die objektorientierte Weiterentwicklung von C ist C++. Andere bekannte objektorientierte Sprachen sind Java, Python, C#, JavaScript, PHP, verschiedene Pascal-Dialekte und viele weitere.

Das Beispiel-Programm `objects-14.cpp` ist eine direkte Übersetzung von `objects-12.c` nach C++. In C++ kümmert sich der Compiler um die Vererbung zwischen den Objekt-Datentypen, um die Verwaltung der Zeiger auf virtuelle Methoden, um korrekte Konstruktoren und um vieles mehr. Auch die Übergabe des Objekt-Zeigers `this` an Methoden erfolgt in C++ automatisch: Aus `object[i]—>base.print (object[i]);` wird `object[i]—>print ();`.

Dadurch daß der Compiler viele Aufgaben übernimmt, die der Programmierer ansonsten manuell abarbeiten müßte, wird der Quelltext kürzer und weniger fehleranfällig.

## 7 Datenstrukturen

### 7.1 Stack und FIFO

Eine häufige Situation beim Programmieren ist, daß man ein Array für eine gewisse Maximalmenge von Einträgen anlegt, aber nur zum Teil nutzt.

Einem derartigen Array ein Element hinzuzufügen, ist einfach: Man erhöht die Variable, die die Auslastung des Arrays speichert. Ebenso einfach ist das Entfernen des zuletzt eingefügten Elements: Man erniedrigt die Variable, die die Auslastung des Arrays speichert.

„Einfach“ bedeutet hier, daß die benötigte Rechenzeit gering ist. Genaugenommen ist die Rechenzeit immer gleich groß, egal wie viele Elemente sich bereits im Array befinden. Die Komplexität (Landau-Symbol) der Operation, am Ende des Arrays ein Element einzufügen oder zu entfernen, ist  $\mathcal{O}(1)$ .

Eine derartige Struktur eignet sich gut, um Elemente in der Reihenfolge des Eintreffens zu speichern, sie aber in *umgekehrter* Reihenfolge wieder abzuarbeiten. Man „stapelt“ gewissermaßen die Elemente in dem Array. Aus diesem Grunde heißt diese Struktur *Stack* (engl.: *Stapel*) oder *LIFO* für *last in, first out*.

Andere Operationen – z. B. das Einfügen oder Löschen von Elementen in der Mitte – sind aufwendiger, da man die im Array befindlichen Elemente in einer Schleife beiseiteschieben muß. Die Rechenzeit ist proportional zur Anzahl der Elemente, die sich bereits im Array befinden:  $\mathcal{O}(n)$ .

Das Suchen in einem bereits sortierten Array ist hingegen in  $\mathcal{O}(\log n)$  möglich: Man beginnt die Suche in der Mitte und prüft, ob sich das gesuchte Element in der unteren oder in der oberen Hälfte befindet. In der ermittelten Hälfte beginnt man die Suche wieder in der Mitte – so lange, bis man nur noch ein einzelnes Element vor sich hat.

Das Beispiel-Programm `stack-11.c` illustriert, wie man einen Stack mit den o. g. Funktionalitäten implementieren kann.

Eine weitere wichtige Situation ist, daß man anfallende Daten zwischenspeichern und *in derselben Reihenfolge* wieder abarbeiten möchte. Eine Struktur, die dies ermöglicht, heißt **FIFO** für *first in, first out*.

Um einen FIFO zu realisieren, verwendet man nicht eine einzelne Variable, die den genutzten Teil des Arrays speichert, sondern zwei: Ein Schreib-Index markiert, an welcher Stelle Platz für das nächste einzufügende Element ist; ein Lese-Index markiert das zuerst eingefügte Element. Beide Indizes werden bei Verwendung hochgezählt. Wenn sie gleich sind, ist der FIFO leer.

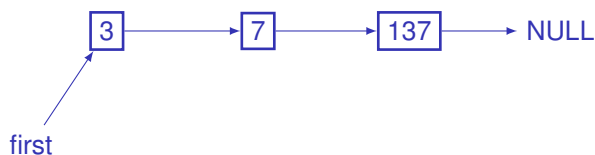
Der entscheidende Trick: Wenn eine der beiden Indexvariablen das Ende des Arrays erreicht, wird sie wieder auf 0 gesetzt. Die beiden Indexvariablen arbeiten also *ringförmig*; der FIFO wird durch einen **Ringpuffer** realisiert.

Beispiel-Programm: [fifo-8.c](#)

## 7.2 Verkettete Listen

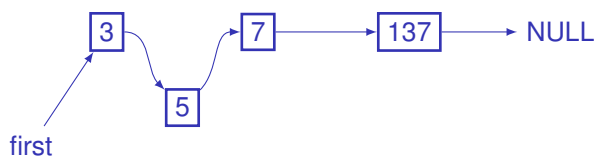
In Arrays ist das Einfügen in der Mitte sehr aufwendig ( $\mathcal{O}(n)$ ). Um das Einfügen zu vereinfachen, hat man sich die folgende Struktur überlegt:

- Jeder Datensatz ist ein **struct**, der zusätzlich zum eigentlichen Inhalt noch einen Zeiger auf das nächste Element enthält.
- Beim letzten Element zeigt der Zeiger auf **NULL**.
- Eine Variable (z. B. **first**) zeigt auf das erste Element.
- Wenn die Liste leer ist, zeigt bereits die **first**-Variable auf **NULL**.



Eine derartige Struktur heißt eine **(einfach) verkettete Liste**.

Wenn nun ein zusätzliches Element in die Liste eingefügt werden soll, geschieht dies durch „Umbiegen“ der Zeiger, die auf das jeweils nächste Element zeigen:



Unabhängig von der Gesamtzahl der Elemente, die sich bereits in der Liste befinden, müssen für das Einfügen eines Elements genau 2 Zeiger neu gesetzt werden. Der Aufwand für das Einfügen beträgt somit  $\mathcal{O}(1)$ .

Diesem Vorteil steht der Nachteil gegenüber, daß es bei einer verketteten Liste nicht mehr möglich ist, „einfach so“ auf das Element mit einem bekannten Index zuzugreifen; dies kann nun nur noch über eine Schleife geschehen. Während bei einem Array der wahlfreie Zugriff in  $\mathcal{O}(1)$  möglich ist, geschieht dies bei einer verketteten Liste in  $\mathcal{O}(n)$ .

Als Konsequenz ist auch das Suchen in  $\mathcal{O}(\log n)$  nicht mehr möglich; auch dies erfordert nun  $\mathcal{O}(n)$ .

## 7.3 Bäume

Für datenintensive Anwendungen – insbesondere Datenbanken – ist es wünschenswert, *sowohl* den wahlfreien Zugriff *als auch* das Einfügen in der Mitte *als auch* das Suchen und das sortierte Einfügen möglichst effizient zu realisieren.

Dies geschieht über rekursive Datenstrukturen, sog. **Bäume**.

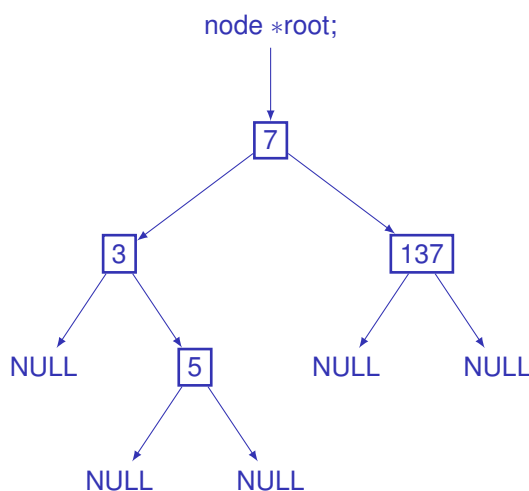
Wie bei einer verketteten Liste sind die Elemente – die *Knoten* – eines Baums **struct**-Variable. Zusätzlich zum eigentlichen Inhalt speichert man darin noch Zeiger auf größere bzw. kleinere Elemente.

Im einfachsten Fall enthält jeder Knoten genau einen Inhalt und jeweils einen Zeiger auf kleinere bzw. größere Elemente:

```
typedef struct node
{
    int content;
    struct node *left, *right;
} node;
```

Ein aus derartigen Knoten aufgebauter Baum verzweigt sich an jedem Knoten in jeweils zwei Teilbäume und heißt daher *binärer Baum*.

Die Struktur ermöglicht es, jeweils „zwischen“ zwei bereits eingefügten Knoten noch weitere einzufügen. Wenn in einen derartigen sortierten binären Baum nacheinander die Zahlen 7, 3, 137 und 5 eingefügt werden, ergibt sich das folgende Bild:



Sowohl das Einfügen als auch die Ausgabe und die Suche erfolgen in Bäumen *rekursiv*. Der Rechenaufwand hängt dabei von der Rekursionstiefe, also von der „Tiefe“ des Baums ab. Da die Tiefe mit der maximal möglichen Anzahl der Knoten logarithmisch wächst, ist Einfügen und Suchen in  $\mathcal{O}(\log n)$  möglich. Dies ist ein Kompromiß zwischen dem Verhalten eines Arrays (Einfügen in  $\mathcal{O}(n)$ , Suchen in  $\mathcal{O}(\log n)$ ) und dem einer verketteten Liste (Einfügen in  $\mathcal{O}(1)$ , Suchen in  $\mathcal{O}(n)$ ). Ein sortiertes Einfügen in einen Baum ermöglicht eine Sortierung in  $\mathcal{O}(n \log n)$ .

Dies funktioniert nur, wenn die Tiefe des Baums tatsächlich nur logarithmisch mit der Anzahl der Knoten wächst. Wenn man in dem oben beschriebenen einfachen Fall eines binären Baums die Elemente in bereits sortierter Reihenfolge einfügt, entartet der Baum zu einer verketteten Liste. Suchen ist dann nur noch in  $\mathcal{O}(n)$  möglich und Sortieren in  $\mathcal{O}(n^2)$ .

Um dies zu vermeiden, wurden teils aufwendige Strategien entwickelt, den Baum jederzeit *balanciert*, d. h. in logarithmischer Tiefe zu halten. Derartige *balancierte Bäume* finden Verwendung in realen Datenbank-Programmen.