Hochschule Bochum
Bochum University
of Applied Sciences

Hardwarenahe Software-Entwicklung

Sommersemester 2012

Prof. Dr. Peter Gerwinski

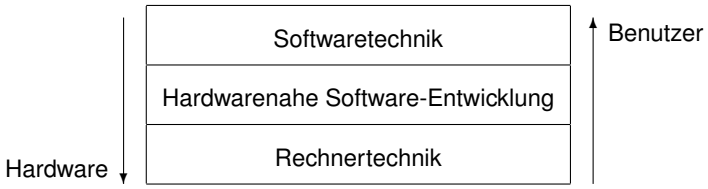
Inhaltsverzeichnis

Stand: 28. Juni 2012

0 Einführung

0.1 Hardwarenahe Software-Entwicklung

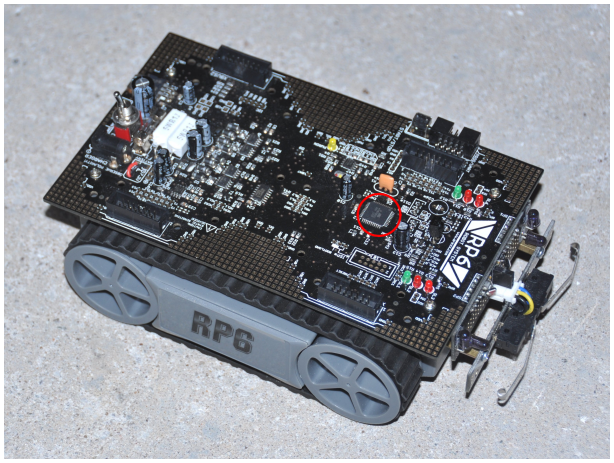
Die hardwarenahe Software-Entwicklung vermittelt zwischen den Fachgebieten der Rechnertechnik („Wie funktioniert ein Computer?“) und der Softwaretechnik (Graphische Benutzeroberflächen, Datenbanken, Software für Endbenutzer, . . .).



Gegenstände der hardwarenahen Software-Entwicklung sind:

- Gerätetreiber
- eingebettete Systeme
- Mikro-Controller

Beispiel: Der unten abgebildete Roboter Robby RP6 (Hersteller: Arexx) ist ein eingebettetes System und wird von einem Mikro-Controller (rot markiert) gesteuert, den man selbst programmieren kann.



0.2 Programmierung in C

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

Warum C?

Wie Sie in Veranstaltungen zum Thema „Rechnertechnik“ noch lernen werden oder bereits gelernt haben, führen die zentralen Recheneinheiten von Computern Befehle aus, die in Gestalt von Zahlen vorliegen. Diese Zahlen heißen *Maschinensprache*. Ihre wörtliche Übersetzung in ein für Menschen lesbares Format heißt *Assemblersprache*.

Die Assemblersprache wäre für hardwarenahe Programmierung die erste Wahl, wenn sie nicht die folgenden Nachteile hätte:

- Programmierung in Assemblersprache ist für Menschen sehr umständlich
  und daher zeitaufwendig und fehleranfällig.
- Maschinen- und Assemblersprache sind für jede *Plattform*,
  d. h. jede Kombination von Hardware und Betriebssystem, völlig unterschiedlich.

Die Programmiersprache C wurde als „High-Level-Assembler“ entwickelt: Sie abstrahiert die Konzepte der Assemblersprache, so daß man plattformunabhängig, aber trotzdem weiterhin hardwarenah programmieren kann.

Im folgenden ist ein Programm, das den Text „Hello, world!“ auf den Bildschirm ausgibt, in den Sprachen C (links), Assembler (Mitte) und Maschinensprache (rechts) wiedergegeben.

```
#include <stdio.h>

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

```
LC0:
.string "Hello, world!\n"
main:
push %ebp
mov %esp,%ebp
and $0xfffffff0,%esp
sub $0x10,%esp
movl $LC0,(%esp)
call printf
mov $0x0,%eax
leave
ret
```

```
48 65 6c 6c 6f 2c 20
77 6f 72 6c 64 21 0a 00

55
89 e5
83 e4 f0
83 ec 10
c7 04 24 00 00 00 00
e8 fc ff ff ff
b8 00 00 00 00
c9
c3
```

Das C-Programm ist auf praktisch jedem Rechner lauffähig, das Assembler- und das Maschinenprogramm jedoch nur auf der konkreten Plattform, für die es jeweils geschrieben wurde (hier: 32-Bit-Linux-PC). Die Zahlen (48 65 6c ...) sind, wie bei Maschinensprache üblich, in hexadezimaler Schreibweise notiert. (Hexadezimalzahlen sind Zahlen zur Basis 16. Zusätzlich zu den üblichen Ziffern 0–9 werden die Buchstaben a–f als weitere Ziffern für die Zahlenwerte 10–15 verwendet. Die hexadezimale Zahlenfolge 48 65 6c entspricht der dezimalen Zahlenfolge 72 101 108. Mehr zu Hexadezimalzahlen finden Sie in Abschnitt 3.1.2.)

Aus diesen Gründen hat sich C als „kleinster gemeinsamer Nenner für viele Plattformen“ als Profi-Werkzeug etabliert.

In dieser Veranstaltung sollen Sie lernen, mit Hilfe der Programmiersprache C die Hardware direkt anzusprechen und effizient einzusetzen.

0.3 Kein „Fallschirm“

Als „High-Level-Assembler“ ist die Programmiersprache C „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 Mikro-Controllern, die im Rahmen dieser Vorlesung 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 könnte, wissen, was Sie tun.

1 Einführung in C

1.1 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 zu einem späteren Zeitpunkt (Abschnitt 2.1) genauer erklärt werden.

- **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 1.8) 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 1.8.)

1.2 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 Maschinensprache 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. `voyager2/home/peter/bo/2012ss/hs/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.
- 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.

1.3 Zahlenwerte ausgeben

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 hello-2.c:

```
#include <stdio.h>

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

Beim Compilieren dieses Programms erhalten wir eine Warnung:

```
$ gcc hello-2.c -o hello-2
hello-2.c: In function 'main':
hello-2.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 hello-2. Wenn wir diese jedoch ausführen, erhalten wir eine Fehlermeldung:

```
$ ./hello-2
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 („Schutzverletzung“) ab.

Auf einer Plattform ohne derartige Schutzmechanismen (z. B. einem Mikro-Controller) 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.

Wie man nun tatsächlich in C Zahlenwerte ausgibt, illustriert das Beispielprogramm hello-3.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 anstelle von %d die Formatspezifikation %x verwendet, wird die Zahl in hexadezimaler anstatt in dezimaler Schreibweise ausgegeben. 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.

Umgekehrt können Sie in C Integer-Konstanten durch Voranstellen von 0x in hexadezimaler anstatt dezimaler Schreibweise eingeben. Die Hexadezimalzahl 0x2a steht in C für genau dieselbe Konstante wie die Dezimalzahl 42.

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-3i.c)

  ```
  #include <stdio.h>

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

  wird vom Compiler akzeptiert. (Warum das so ist, wird in Abschnitt 1.7 behandelt.)

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

  ```
  $ gcc -Wall hello-3i.c -o hello-3i
  hello-3i.c: In function 'main':
  hello-3i.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: hello-3j.c):

```
#include <stdio.h>

int main (void)
{
  printf ("%d␣plus␣%d␣ist␣%d\n", 2, 2, 4);
  return 0;
}
```

```
$ gcc hello-3j.c -o hello-3j
$ ./hello-3j
2 plus 2 ist 4
```

Achtung: Zu viele oder zu wenige Werte in der Parameterliste ergeben trotzdem ein gültiges, wenn auch fehlerhaftes C-Programm (Datei: hello-3k.c):

```
#include <stdio.h>

int main (void)
{
  printf ("%d␣plus␣%d␣ist␣%d\n", 2, 2);
  return 0;
}
```

Wenn man dieses Programm laufen läßt, werden für den fehlenden Parameter Werte genommen, die sich zufällig dort im Speicher befinden, wo der Wert hätte stehen sollen:

```
$ gcc hello-3k.c -o hello-3k
$ ./hello-3k
2 plus 2 ist -1216061452
```

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

```
$ gcc -Wall hello-3k.c -o hello-3k
hello-3k.c: In function 'main':
hello-3k.c:5: warning: too few arguments for format
```

1.4 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:

+	Addition
−	Subtraktion
∗	Multiplikation
/	Division: Bei ganzen Zahlen wird grundsätzlich abgerundet.
%	Rest bei Division (39 % 4 ergibt 3.)

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.

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

Das Beispielprogramm mathe-2.c liest einen Wert vom Standardeingabegerät (hier: Tastatur) ein und gibt das Doppelte des Wertes auf dem Standardausgabegerät aus:

```
#include <stdio.h>

int main (void)
{
  int a;
  scanf ("%d", &a);
  printf ("%d\n", 2 * 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 1.9 genauer behandelt.)

Wenn wir das & vergessen (Beispielprogramm: mathe-2a.c), kann das C-Programm weiterhin compiliert 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.)

Es ist sinnvoll, dem Benutzer mitzuteilen, wie das Programm benutzt werden soll. In diesem Fall ist es zweckmäßig, ihn zur Eingabe einer Zahl aufzufordern.

Die Funktion scanf() dient ausschließlich dazu, Werte einzulesen. Es ist in scanf() selbst nicht vorgesehen, Benutzerhinweise auszugeben. Hierfür verwenden wir stattdessen wieder die Funktion printf().

Beispielprogramm: mathe-2b.c

```
#include <stdio.h>

int main (void)
{
  int a;
  printf ("Bitte␣eine␣Zahl␣eingeben:␣");
  scanf ("%d", &a);
  printf ("Das␣Doppelte␣ist:␣%d\n", 2 * a);
  return 0;
}
```

Um mit Fließkomma- anstelle von ganzen Zahlen zu rechnen, ersetzen wir den Datentyp **int** durch **float** und die Formatspezifikation %d durch %f.

Beispielprogramm: mathe-3.c

```
#include <stdio.h>

int main (void)
{
  float a;
  printf ("Bitte␣eine␣Zahl␣eingeben:␣");
  scanf ("%f", &a);
  printf ("Das␣Doppelte␣ist:␣%f\n", 2 * a);
  return 0;
}
```

Wie im englischen Sprachraum üblich, wird auch in C als Dezimaltrennzeichen ein Punkt (und kein Komma) verwendet.

Der Datentyp **float** hat eine eher geringe Genauigkeit von typischerweise 7–8 Dezimalstellen (4-Byte-Zahl im IEEE-754-Standard). Wenn eine höhere Genauigkeit benötigt wird, verwendet man anstelle von **float** den Datentyp **double** und anstelle von %f die Formatspezifikation %lf (mit „l“ wie „long“, Beispielprogramm: mathe-3a.c).

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.

Dies wird im folgenden Beispielprogramm mathe-3b.c illustriert:

```c
#include <stdio.h>

int main (void)
{
  double a;
  printf ("Bitte␣eine␣Zahl␣eingeben:␣");
  scanf ("%lf", &a);
  a = 2 * a;
  printf ("Das␣Doppelte␣der␣Zahl␣ist:␣%lf\n", a);
  return 0;
}
```

Die Zeile 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, das dann wiederum in der Variablen a gespeichert wird.

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

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

Auf welche Weise die vom Benutzer eingegebenen Zeichen in Werte für die Variablen umgesetzt werden, ist in der Dokumentation zu scanf() beschrieben. (In der Unix-Shell können Sie diese mit dem Befehl `man 3 scanf` abrufen.)

1.5 Verzweigungen

Das Beispielprogramm if-0.c

```c
#include <stdio.h>

int main (void)
{
  int a, b;
  printf ("Bitte␣erste␣Zahl␣eingeben:␣");
  scanf ("%d", &a);
  printf ("Bitte␣zweite␣Zahl␣eingeben:␣");
  scanf ("%d", &b);
  printf ("Die␣erste␣Zahl␣geteilt␣durch␣die␣zweite␣ergibt:␣%d,␣Rest␣%d␣\n", a / b, 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 erste Zahl eingeben: 13
Bitte zweite Zahl 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 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␣erste␣Zahl␣eingeben:␣");
  scanf ("%d", &a);
  printf ("Bitte␣zweite␣Zahl␣eingeben:␣");
  scanf ("%d", &b);
  if (b != 0)
    printf ("Die␣erste␣Zahl␣geteilt␣durch␣die␣zweite␣ergibt:␣%d,␣Rest␣%d␣\n", a / b, 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␣erste␣Zahl␣eingeben:␣");
  scanf ("%d", &a);
  printf ("Bitte␣zweite␣Zahl␣eingeben:␣");
  scanf ("%d", &b);
  if (b != 0)
    printf ("Die␣erste␣Zahl␣geteilt␣durch␣die␣zweite␣ergibt:␣%d,␣Rest␣%d␣\n", a / b, 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␣erste␣Zahl␣eingeben:␣");
  scanf ("%d", &a);
  printf ("Bitte␣zweite␣Zahl␣eingeben:␣");
  scanf ("%d", &b);
  if (b != 0)
    printf ("Die␣erste␣Zahl␣geteilt␣durch␣die␣zweite␣ergibt:␣");
    printf ("%d,␣Rest␣%d␣\n", a / b, a % b);
  return 0;
}
```

wird die Zeile printf ("%d,␣Rest␣%d␣\n", a / b, a % b); auch dann ausgeführt, wenn die Variable b den Wert 0 hat.

Wenn hinter dem letzten printf noch ein **else** stünde, wäre das Programm aufgrund derselben Regel kein korrektes C mehr.

In C ist die Einrückung der Zeilen im Programmquelltext „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 geschweifter Klammern zu einem sog. *Anweisungsblock* zusammenfassen.

Beispielprogramm: if-4.c

```
#include <stdio.h>

int main (void)
{
  int a, b;
  printf ("Bitte␣erste␣Zahl␣eingeben:␣");
  scanf ("%d", &a);
  printf ("Bitte␣zweite␣Zahl␣eingeben:␣");
  scanf ("%d", &b);
  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");
  return 0;
}
```

Aus Sicht des Computers ist die Einrückung 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␣erste␣Zahl␣eingeben:␣");scanf("%d",&
a);printf("Bitte␣zweite␣Zahl␣eingeben:␣");scanf("%d",&b);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");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 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. Beispielsweise gibt das folgende Programm bool-1.c den Zahlenwert 0 aus:

```
#include <stdio.h>

int main (void)
{
  int a = 7;
  printf ("%d\n", a == 8);
  return 0;
}
```

1.6 Schleifen

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

Das folgende Beispielprogramm while-1.c berechnet alle Zweierpotenzen, die kleiner als 1000 sind:

```
#include <stdio.h>

int main (void)
{
  int a;
  a = 1;
  while (a < 1000)
    {
      a = a + a;
      printf ("%d\n", a);
    }
  return 0;
}
```

Die Auswertung der Bedingung erfolgt analog zur **if**-Anweisung. Ebenso folgt auf **while** nur eine 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 als der rechte ist, ansonsten 0. Entsprechend sind die Operatoren >, <= und >= definiert.

Wenn man eine Bedingung angibt, die niemals 0 wird, erzeugt man eine Endlosschleife (while-2.c):

```c
#include <stdio.h>

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

Das endlos laufende Programm kann nur noch über das Betriebssystem beendet werden. Von der Unix-Shell aus geschieht dies durch Eingabe von `Strg+C`.

In der Ausgabe des oben dargestellten Beispielprogramms fällt auf, daß die Zweierpotenzen zunächst wie erwartet anwachsen, später aber nur noch der Zahlenwert 0 ausgegeben wird:

```
$ gcc -Wall while-2.c -o while-2
$ ./while-2
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432
67108864
134217728
268435456
536870912
1073741824
-2147483648
0
0
0
...
```

Dies hängt mit der Art und Weise zusammen, wie Zahlen in einem Computer gespeichert werden. Im Detail ist dies Gegenstand der Vorlesung „Rechnertechnik“; ein paar für uns wichtige Eckdaten seien an dieser Stelle erwähnt:

- Computer speichern Zahlen im Binärformat, das nur die Ziffern 0 und 1 kennt.
  Beispielsweise lautet die Dezimalzahl 9 in Binärdarstellung 1001.
- Zweierpotenzen entsprechen jeweils einer 1 mit folgenden Nullen.
  Die Dezimalzahlen 2, 4, 8 und 16 haben die Binärdarstellungen 10, 100, 1000 und 10000.
- Auf einem 32-Bit-Prozessor zeigt die zweiundreißigste Ziffer von rechts das Vorzeichen an. Steht hier eine 1, ist die Zahl negativ. Dies erklärt, weshalb die Verdopplung von 1073741824 (binär: eine 1 mit 30 Nullen) -2147483648 ergibt (binär: eine 1 mit 31 Nullen).
- Bei einer weiteren Verdopplung würde binär eine 1 mit 32 Nullen entstehen, die aber von einem 32-Bit-Prozessor nicht mehr dargestellt werden kann. Ohne weitere Maßnahmen ist daher das Doppelte von -2147483648 auf einem 32-Bit-Prozessor die Zahl 0.

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

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

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

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

ist genau dasselbe wie

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

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

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

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

zwar zulässiges C, aber nicht sinnvoll. Dies kann man sofort erkennen, indem man die **for**-Schleife in eine **while**-Schleife übersetzt:

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

Dieses Programmfragment setzt einmalig `a` auf den Wert 10 und springt danach in eine Endlosschleife zur Ausgabe von `a`. (Die **while**-Bedingung `10` ist ungleich Null, hat also stets den Wahrheitswert „wahr“.) Am Ende jedes Schleifendurchlaufs wird *nichts* gemacht, weil innerhalb der Klammern des **for** rechts vom zweiten Semikolon nichts mehr stand.

1.7 Seiteneffekte

Wie wir anhand des Beispielprogramms hello-3i.c

```
#include <stdio.h>

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

gesehen haben, akzeptiert C auch sehr seltsame Konstrukte.

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

Dies erklärt die Bedeutung der (gültigen!) C-Anweisung "\n";: „Berechne den Wert "\n" und vergiß ihn wieder."

Tatsächlich ist printf ("\n%d", 42); eine Anweisung genau gleichen Typs: 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 side-effects-1.c gibt den vom ersten printf() zurückgegebenen Wert mit Hilfe eines zweiten printf() aus:

```
#include <stdio.h>

int main (void)
{
  int a = printf ("%d\n", 42);
  printf ("%d\n", a);
  return 0;
}
```

Die Ausgabe lautet:

```
42
3
```

Bei dem zurückgegebenen Wert handelt es sich um die Anzahl der geschriebenen Zeichen. In diesem Fall sind es zwei Ziffern plus 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 Operatoren an, weisen als Seiteneffekt das Ergebnis dem ersten Operanden zu und geben das Rechenergebnis als Wert zurück.
- Die binären Rechnoperatoren + − ∗ / % 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.
- 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.

1.8 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() und wird nur zu Programmbeginn initialisiert.

  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.

1.9 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.

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.

1.10 Arrays und Strings

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, i = 0;
  while (i < 5)
    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 i = 0;
```

```
  while (i < 5)
    printf ("%d\n", prime[i++]);
  return 0;
}
```

(Zur Erinnerung: Das ++ in prime[i++] besagt, daß wir prime[i] an printf() übergeben und *danach* den Index i um 1 erhöhen.)

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 **char**s, also ein Zeiger auf **char**s, 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 arrays-3--.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 Verwendung von Pointer-Arithmetik und Weglassen der überflüssigen Abfrage != 0 erhalten wir das äquivalente Beispielprogramm arrays-3.c:

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello,␣world!\n";
  char *p = hello_world;
  while (*p)
    printf ("%d", *p++);
  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: arrays-3a.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: arrays-4.c

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello, world!\n";
  printf ("%s", hello_world);
  return 0;
}
```

Anstatt als Array können wir die Variable hello_world auch als Zeiger deklarieren (Datei: arrays-4a.c):

```
#include <stdio.h>

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

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 ausgegegeben.
%c	Der Parameter wird als Zahlenwert interpretiert und als Zeichen ausgegeben.
%s	Der Parameter wird als Zeiger interpretiert und als Zeichenfolge ausgegeben.

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: args-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] = \"%s\"\n", 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, also ein Array von Arrays von **char**s, also ein Array von Strings. Wenn wir es mit einem Index i versehen, greifen wir auf 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 args-1.c -o args-1
$ ./args-1 foo bar baz
argc = 4
argv[0] = "./args-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) oder die Schleife mit dem Erreichen der Endmarkierung abbrechen (Schleifenbedingung argv[i] != NULL).

```
#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] = \"%s\"\n", 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: args-3.c):

```
#include <stdio.h>

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

1.11 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 = { 11, 4, 2012 };
  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 = 11;
  (*d).month = 4;
  (*d).year = 2012;
}

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 = 11;
  d->month = 4;
  d->year = 2012;
}

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: incdate-1.c):

```
#include <stdio.h>

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

void get_date (date *d)
{
  d->day = 11;
  d->month = 4;
  d->year = 2012;
}
```

```
int main (void)
{
  date today;
  get_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() (Datei: incdate-2.c):

```
#include <stdio.h>

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

void get_date (date *d)
{
  d->day = 11;
  d->month = 4;
  d->year = 2012;
}

void inc_date (date *d)
{
  d->day = 11;
  d->month = 4;
  d->year = 2012;
}

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

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-3.c)

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

Diese naive Vorgehensweise versagt, sobald wir den Tag über das Ende des Monats hinauszählen (Beispielprogramm: incdate-4.c). Dies reparieren wir im nächsten Schritt, wobei wir für den Moment inkorrekterweise annehmen, daß alle Monate 30 Tage hätten (Datei: incdate-5.c):

```
void inc_date (date *d)
{
  d->day += 1;
  if (d->day > 30)
    {
      d->day = 1;
      d->month += 1;
      if (d->month > 12)
        {
          d->month = 1;
          d->year += 1;
        }
    }
}
```

Diese Version versagt in Monaten mit mehr oder weniger als 30 Tagen (Beispielprogramm: incdate-6.c).

Die Reparatur nehmen wir wieder stufenweise vor. Zunächst lagern wir die Konstante 30 in eine Funktion aus (Datei: incdate-7.c)

```
int days_in_month (date *d)
{
  return 30;
}

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

Anschließend „reparieren“ wir die fehlerhafte Funktion, wobei wir zunächst das Problem der Schaltjahre aussparen (Datei: incdate-8.c):

```
int days_in_month (date *d)
{
  if (d->month == 4)
    return 30;
  else if (d->month == 6)
    return 30;
  else if (d->month == 9)
    return 30;
  else if (d->month == 11)
    return 30;
  else if (d->month == 2)
    return 28;
  else
    return 31;
}
```

Dieses Problem (Datei: incdate-9.c) reparieren wir durch Auslagern des Problems „Schaltjahr oder nicht?“ in eine zunächst fehlende (Datei: incdate-10.c) und danach fehlerhafte Funktion (Datei: incdate-11.c):

```
int is_leap_year (date *d)
{
  if (d->year % 4 == 0)
    return 1;
  else
    return 0;
}

int days_in_month (date *d)
{
  if (d->month == 4)
    return 30;
  else if (d->month == 6)
    return 30;
  else if (d->month == 9)
    return 30;
  else if (d->month == 11)
    return 30;
  else if (d->month == 2)
    {
      if (is_leap_year (d))
        return 29;
      else
        return 28;
    }
  else
    return 31;
}
```

Das nun vorliegende Programm arbeitet bereits für den julianischen Kalender sowie für alle Jahre von 1901 bis 2099 korrekt. Um durch 100 teilbare Jahre korrekt als Nicht-Schaltjahre zu erfassen (Datei: incdate-12.c), ergänzen wir nun die 100-Jahre-Regel des gregorianischen Kalenders (Datei: incdate-13.c):

```
int is_leap_year (date *d)
{
  if (d->year % 4 == 0)
    {
      if (d->year % 100 == 0)
        return 0;
      else
        return 1;
    }
  else
    return 0;
}
```

Um schließlich auch durch 400 teilbare Jahre korrekt als Schaltjahre zu erfassen (Datei: incdate-14.c) ergänzen wir nun als letztes die 400-Jahre-Regel des gregorianischen Kalenders. Damit ist die Aufgabe gelöst. Der vollständige Quelltext der Lösung (Datei: incdate-15.c) lautet:

```
#include <stdio.h>

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

void get_date (date *d)
{
  d->day = 28;
  d->month = 2;
```

```
  d->year = 2000;
}

int is_leap_year (date *d)
{
  if (d->year % 4 == 0)
    {
      if (d->year % 100 == 0)
        {
          if (d->year % 400 == 0)
            return 1;
          else
            return 0;
        }
      else
        return 1;
    }
  else
    return 0;
}

int days_in_month (date *d)
{
  if (d->month == 4)
    return 30;
  else if (d->month == 6)
    return 30;
  else if (d->month == 9)
    return 30;
  else if (d->month == 11)
    return 30;
  else if (d->month == 2)
    {
      if (is_leap_year (d))
        return 29;
      else
        return 28;
    }
  else
    return 31;
}

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

int main (void)
{
  date today;
  get_date (&today);
```

```
  inc_date (&today);
  printf ("%d.%d.%d\n", today.day, today.month, today.year);
  return 0;
}
```

Bemerkungen:

- 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 uncompilierbarer Symbole haben sich hier Kommentare wie */*FIXME */* bewährt, auf die man seinen Code vor der Auslieferung der Endfassung noch einmal automatisch durchsuchen läßt.
- Allen an der Berechnung beteiligten Funktionen wurde hier ein Zeiger d auf die vollständige date-Struktur übergeben. Dies ist ein *objektorientierter Ansatz*, bei dem man die Funktionen als *Methoden* der *Klasse* date auffaßt. (Von sich aus unterstützt die Sprache C – im Gegensatz zu z. B. C++ – keine Klassen und Methoden, sondern man muß diese bei Bedarf in der oben beschrieben Weise selbst basteln. Für eine fertige Lösung siehe z. B. die GObject-Bibliothek – http://gtk.org.)

  Alternativ könnte man sich mit den zu übergebenden Parametern auf diejenigen beschränken, die in der Funktion tatsächlich benötigt werden, also z. B. **int** days_in_month (**int** month, **int** year) und **int** is_leap_year (**int** year). Damit wären die Funktionen allgemeiner verwendbar.

  Welcher dieser beiden Ansätze der bessere ist, hängt von der Situation und von persönlichen Vorlieben ab.

2 Bibliotheken

2.1 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:

```
Es war 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

```
Es war 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"
Es war 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 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 1000 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.

2.2 Bibliotheken einbinden

Tatsächlich ist von den über 1000 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 ("Drei mal vier = %d\n", 3 * 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 VIER 2 + 2

int main (void)
{
  printf ("Drei␣mal␣vier␣=␣%d\n", 3 * VIER);
  return 0;
}
```

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

```
  printf ("Drei␣mal␣vier␣=␣%d\n", 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 enstanden ist.

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

```
#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.)

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 linken. 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.

2.3 Bibliotheken verwenden (Beispiel: OpenGL)

Die *OpenGL*-Bibilothek 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.)

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

- Funktionen: glutInit (&argc, argv);
- Konstante: GLUT_RGBA
- Datentypen: 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. (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);   <-- Bildschirm löschen
  set_material_color (1.0, 1.0, 0.0);   <-- Rot- und Grünanteil 100 %, Blauanteil 0 %
  glutSolidCube (0.5);   <-- Würfel zeichnen
  glFlush ();   <-- Zeichnung „abschicken“
}

int main (int argc, char **argv)
{
  init_opengl (&argc, argv, "Cube");
  glutDisplayFunc (draw);   <-- Callback-Funktion installieren (s. o.)
  glutMainLoop ();   <-- Endlosschleife: Ab jetzt werden nur noch Callbacks aufgerufen.
  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.

- Weil das Bild während des Neuzeichnens die ganze Zeit zu sehen ist, flackert in cube-3.c der Bildschirm. Dies wird in cube-3a.c dadurch behoben, daß die Zeichnung zunächst in einem unsichtbaren Pufferspeicher aufgebaut wird. Erst die fertige Zeichnung wird mit dem Funktionsaufruf swapBuffers() sichtbar gemacht.

Damit dies möglich ist, muß beim Initialisieren ein doppelter Puffer angefordert werden. Zu diesem Zweck ersetzen wir die Bibliothek opengl-magic.c durch opengl-magic-double.c.

(Dies illustriert, daß der Include-Mechanismus des Präprozessors und der Zusammenbau-Mechanismus des Linkers tatsächlich voneinander unabhängig sind.)

(Durch das Austauschen von Bibliotheken, insbesondere bei dynamischen Bibliotheken (Endung .so unter Unix bzw. .dll unter Microsoft Windows) ist es möglich, das Verhalten bereits fertiger Programme zu beeinflussen, ohne das Programm neu compilieren zu müssen. Dies kann zu Testzwecken geschehen, zur Erweiterung des Funktionsumfangs oder auch zum Einschleusen von Schadfunktionen.)

- In cube-3a.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 während des Aufrufs von glPushMatrix() befand.

  Dies ist in cube-3b.c (langsame Drehung) und cube-3c.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.

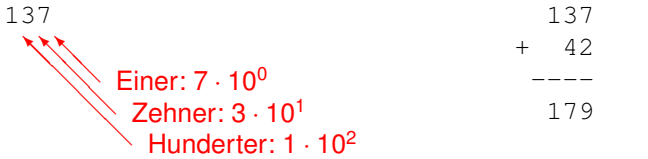
- Die Beispielprogramme orbit-1.c und orbit-1a.c illustrieren eine weitere Transformation der gezeichneten Objekte, nämlich die Translation (Verschiebung). Jeder Transformationsbefehl wirkt sich jeweils auf die *danach* erfolgenden Zeichenbefehle aus. Um sich zu veranschaulichen, welche Transformationen auf ein gezeichnetes Objekt wirken (hier z. B. auf glutSolidSphere()), muß man die Transformationen in der Reihenfolge *von unten nach oben* ausführen.
- Die Beispielprogramme orbit-2.c und orbit-3.c verwenden weitere Transformationen und geometrische Objekte, um die Umlaufbahn zweier Kugeln, beispielsweise des Mondes um die Erde, zu illustrieren.
- Das Beispielprogramm orbit-x.c schließlich versieht 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 Grundlagen hardwarenaher Programmierung

3.1 Zahlensysteme

3.1.1 Dezimalsystem

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



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

3.1.2 Hexadezimalsystem

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

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

- Schreibweise in C: `0x137`

3.1.3 Oktalsystem

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

```
137                                  137
                                   +  42
                                    --11-
                                     201
```

$7 \cdot 8^0$
$3 \cdot 8^1$
$1 \cdot 8^2$

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

- Schreibweise in C: `0137`
- Rechner für beliebige Zahlensysteme: GNU bc

```
$ bc
ibase=8
137
95
obase=10
137 + 42
201
```

Eingabe zur Basis 8
Ausgabe zur Basis 10
Eingabe zur Basis 8 ($10_8 = 8$)
Ausgabe zur Basis 8

3.1.4 Binärsystem

- Basis: 2
- Gültige Ziffern: 0, 1

```
110                                   110
                                    + 1100
                                     --11---
                                     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
001  1
010  2
011  3
100  4
101  5
110  6
111  7
```

```
0000  0
0001  1
0010  2
0011  3
0100  4
0101  5
0110  6
0111  7
1000  8
1001  9
1010  A
1011  B
1100  C
1101  D
1110  E
1111  F
```

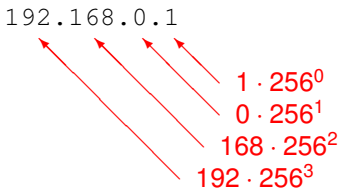
Beispiel: Oktal-Schreibweise für Unix-Zugriffsrechte

`-rw-r-----` = $0\,110\,100\,000_2 = 640_8$ `$ chmod 640 file.c`

`-rwxr-x---` = $0\,111\,101\,000_2 = 750_8$ `$ chmod 750 subdir`

3.1.5 IP-Adressen (IPv4)

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



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

3.2 Bit-Operationen

```
  0110       0110       0110       0110                  0110
+ 1100     | 1100     & 1100     ^ 1100     ~ 1100   >>    2
 -----      -----      -----      -----      -----      -----
 10010      1110       0100       1010       0011       0001
```

Addition | Oder | Und | Exklusiv-Oder | Negation | Bit-Verschiebung

```
  01101100           01101100           01101100
| 00000010         & 11110111         ^ 00010000
 ---------          ---------          ---------
  01101110           01100100           01111100
```

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. `6 & 12 == 4`
  Die logischen Operatoren (z. B. && in C) prüfen die Zahl insgesamt auf $\neq 0$. `6 && 12 == 1`
  Nicht verwechseln!

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

```
  00000001           01101100
<<        2        | 00000100
  ---------          ---------
  00000100           01101100
```

Maske für Bit 2 | Bit gezielt setzen

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

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

```
   00000001                          01101100
<<        2          ~ 00000100   & 11111011
  ---------           ---------     ---------
   00000100            11111011      01101000
```

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:

```
   00000001            01101100          00000100
<<        2          & 00000100       >>        2
  ---------           ---------         ---------
   00000100            00000100          00000001
```

Maske für Bit 2 — Bit 2 isolieren — in Zahl 0 oder 1 umwandeln

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

Beispiel: Netzmaske für 256 IP-Adressen

```
  192.168.  1.123   <-- IP-Adresse eines Rechners
& 255.255.255.  0   <-- Netzmaske: 255 = 11111111(2)
 ----------------
  192.168.  1.  0   <-- IP-Adresse des Sub-Netzes
```

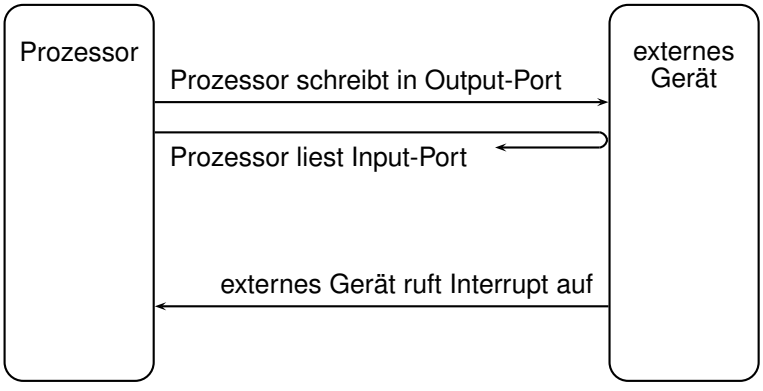
Beispiel: Netzmaske für 8 IP-Adressen

```
  192.168.  1.123   <-- IP-Adresse eines Rechners        01111011
& 255.255.255.248   <-- Netzmaske                     & 11111000
 ----------------                                     ----------
  192.168.  1.120   <-- IP-Adresse des Sub-Netzes      01111000
```

3.3 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.

Beispiel für einen AVR-Mikro-Controller:

```
#include <avr/io.h>

#define F_CPU 8000000
#include <util/delay.h>

int main (void)
{
  DDRC = 0x10;
  while (1)
    {
      PORTC ^= 0x10;
      _delay_ms (500);
    }
  return 0;
}
```

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

Beispiel:

```
#include <avr/io.h>

int main (void)
{
  DDRC = 0x10;
  while (1)
    {
      while ((PINC & 0x01) == 0)
        ; /* just wait */
      PORTC ^= 0x10;
      while ((PINC & 0x01) != 0)
        ; /* just wait */
    }
  return 0;
}
```

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.

Die direkte Ansteuerung von I/O-Ports ist nur auf Mikro-Controllern 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.

3.4 Interrupts

Ein Interrupt ist ein Unterprogramm, das nicht durch einen Befehl (Funktionsaufruf), 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 einen AVR-Mikro-Controller: Hier ist das Gerät, das den Interrupt auslöst, eine Uhr. Über Output-Ports (TCCR0, TIMSK) teilen wir der Uhr mit, wie oft sie den Interrupt auslösen soll. Während dieser Zeit müssen Interrupts gesperrt werden (cli(), sei()).

```
#include <avr/interrupt.h>

ISR (TIMER0_COMP_vect)
{
  PORTC = 0xff;
}

int main (void)
{
  cli ();
  TCCR0 = (1 << CS01) | (1 << CS00);
  TIMSK = 1 << OCIE0;
  sei ();
  while (1)
    ; /* do nothing */
  return 0;
}
```

- 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 `TIMER0_COMP_vect` teilen wir dem Compiler mit, daß er den Timer-Interrupt-Vektor Nr. 0 (also den ersten) auf diesen Interrupt-Handler zeigen lassen soll.

(Tatsächlich handelt es sich bei ISR und INT0_vect um Präprozessor-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.

4 Algorithmen

4.1 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.:

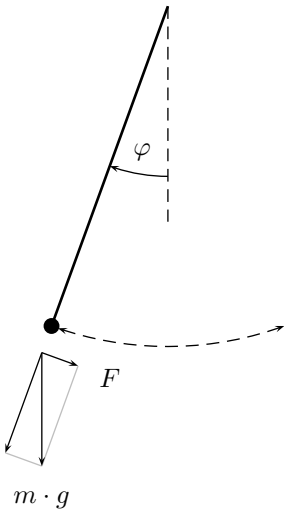
$$x'(t) = -x(t) \qquad \text{mit} \qquad x(0) = 1$$

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.

Beispiel: Pendelschwingung

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



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

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

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

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

lösen.

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

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

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

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

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

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 Δt. 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 Δt erhöhen, erhalten wir Näherungen für die Funktionen $x(t)$ und $v(t)$.

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

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

Mit Hilfe dieser – innerhalb des Zeitintervalls näherungsweise konstanten – Beschleunigung berechnen wir die neue tangentiale 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 φ, wobei wir einen kleinen Umweg über den Kreisbogen $x = l \cdot \varphi$ machen:

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

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 $\sin \varphi \approx \varphi$ 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.

4.2 Ganzzahl-Arithmetik

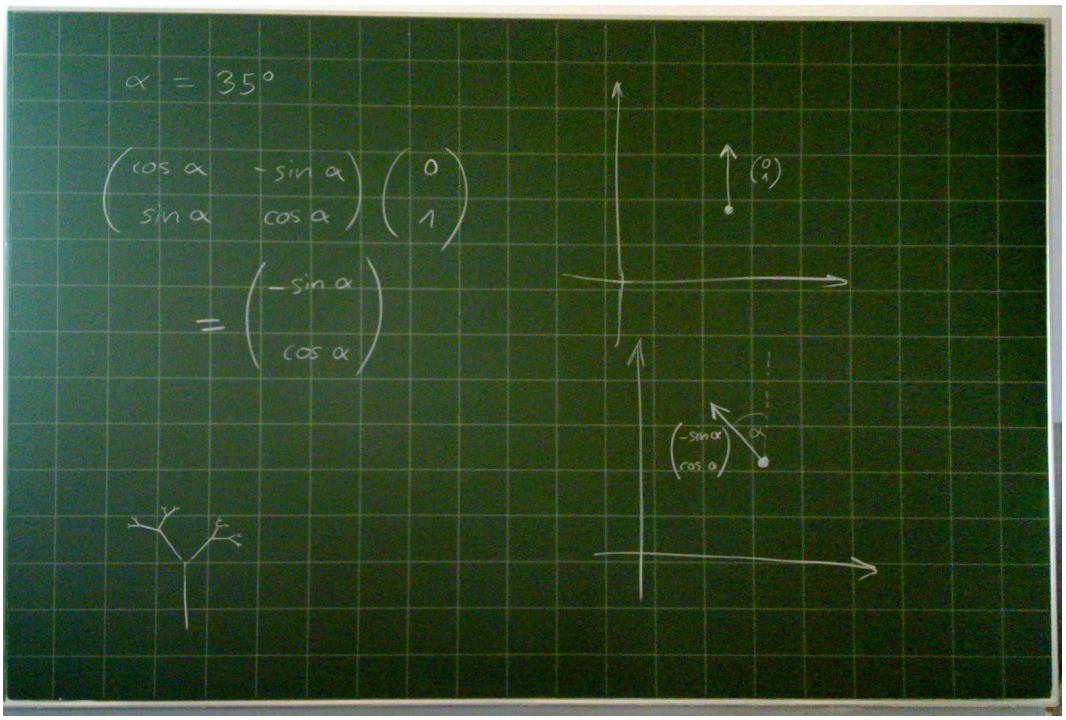
Siehe die Beispielprogramme line∗.c

4.3 Rekursion

Beispiele:

- Türme von Hanoi
  Siehe die Vortragsfolien hs-20120524.pdf
- Floodfill-Algorithmus
  Siehe die Beispielprogramme fill∗.c

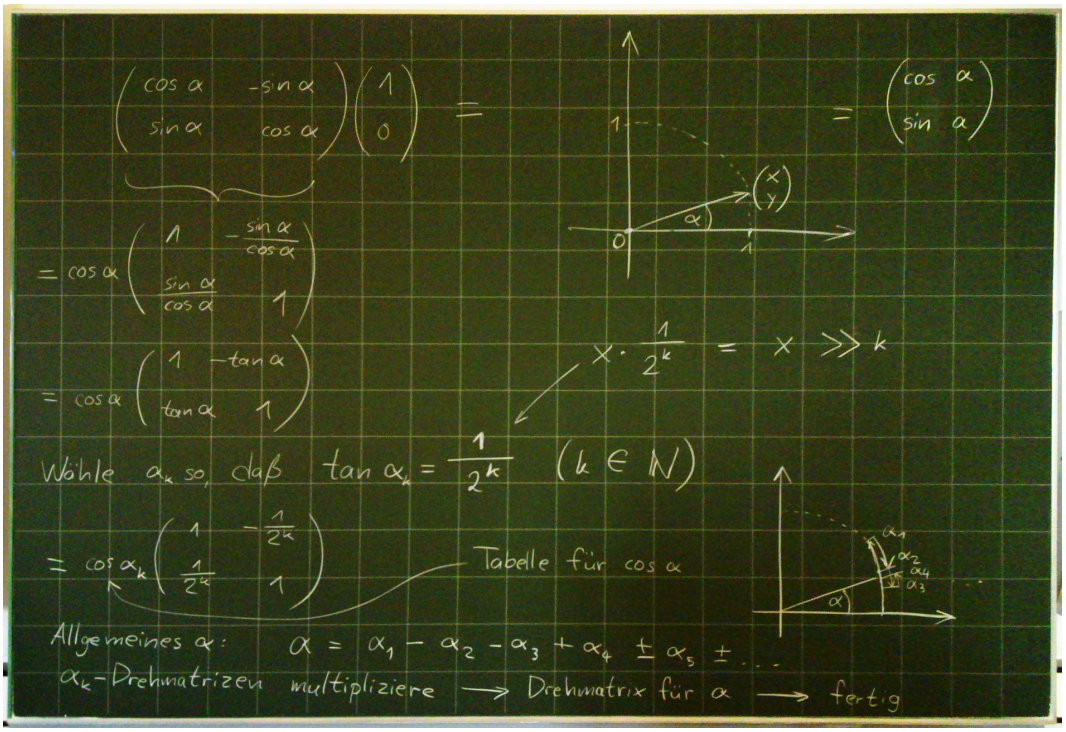
4.4 Drehmatrizen



Rekursion und Drehmatrizen:
Siehe die Beispielprogramme plant∗.c

4.5 CORDIC

- Algorithmus zur effizienten Berechnung von (u. a.) Sinus und Cosinus
- in Hardware-Schaltungen realisierbar



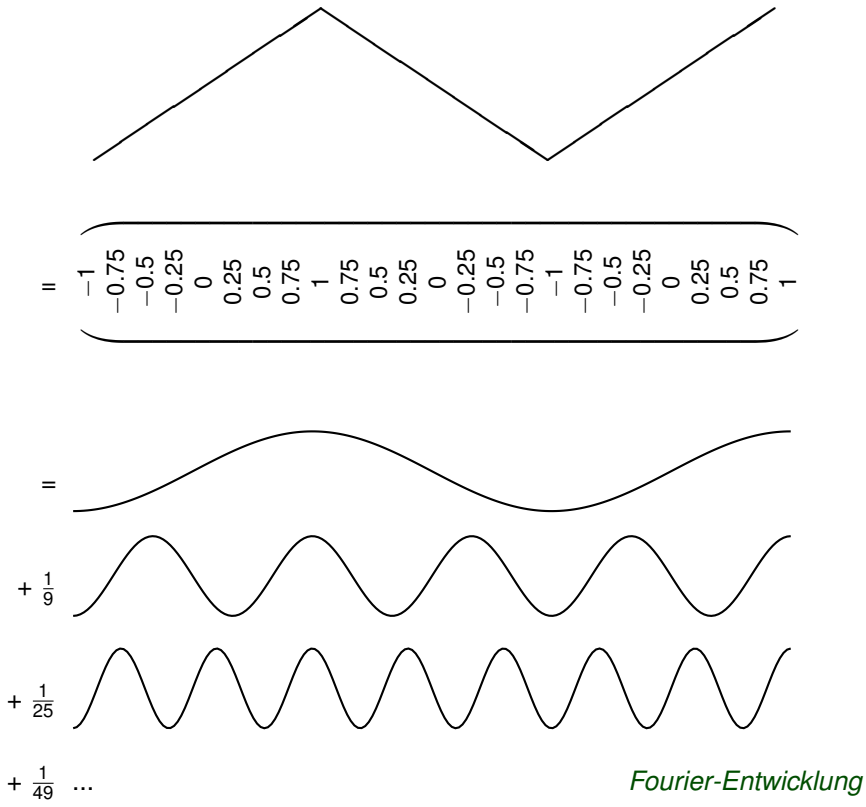
4.6 FFT

Verschiedene Basen für denselben Vektor

Vektoren in $\mathbb{R}^3$:

$$2\begin{pmatrix}1\\0\\1\end{pmatrix}+3\begin{pmatrix}0\\1\\-1\end{pmatrix}+5\begin{pmatrix}-1\\1\\0\end{pmatrix}=\begin{pmatrix}-3\\8\\-1\end{pmatrix}$$
$$=-3\begin{pmatrix}1\\0\\0\end{pmatrix}+8\begin{pmatrix}0\\1\\0\end{pmatrix}-1\begin{pmatrix}0\\0\\1\end{pmatrix}$$

Funktion als Vektor:



Fourier-Entwicklung

4.7 Effizient programmieren

4.7.1 Manuelle und automatische Optimierung

- **Multiplikationen und Divisionen sind teuer:** Bit-Verschiebungen (`b = a >> 2`) bewirken dasselbe wie Multiplikationen mit bzw. Divisionen durch Zweierpotenzen (`b = a / 4`).

  `gcc` mit Optimierung (`-O`, `-O2`, `-O3`) bewirkt dies automatisch.

  $\longrightarrow$ So hinschreiben, daß es möglichst gut dokumentiert, was man vorhat

- **Schleifen sind teuer:** Wenn man etwas vorab außerhalb einer Schleife berechnen kann, sollte man das tun.

  `gcc` mit Optimierung (`-O`, `-O2`, `-O3`) bewirkt dies teilweise automatisch.

- **Manchmal lohnt es sich, Schleifen zu entrollen, manchmal nicht:** Bei Schleifen bekannter Länge ist die Verwaltung der Laufvariablen manchmal „teurer“ als die eigentliche „Nutzlast“ der Schleife.

  ```
  c[0] = a[0] + b[0];
  c[1] = a[1] + b[1];
  c[2] = a[2] + b[2];
  ```
  oder
  ```
  for (int i = 0; i < 3; i++)
    c[i] = a[i] + b[i];
  ```
  ?

  Was besser ist, hängt von der Situation ab.

  `gcc` kann beides automatisch optimieren:
  `gcc -O3`: möglichst schneller Code
  `gcc -Os`: möglichst platzsparender Code (für Micro-Controller)

4.7.2 Programmiertips

- **Einrückung:** Ordnen Sie Ihren Quelltext möglichst übersichtlich an. Eine korrekte Einrückung hilft enorm, den Überblick über das Programm zu behalten, z. B. wenn es notwendig wird, eine vor mehreren Jahren geschriebene Stelle noch einmal zu überarbeiten.

  Dies gilt in noch stärkerem Maße, wenn mehrere Programmierer gemeinsam an einem Programm arbeiten.

- **Top-Down-Ansatz:** Der oben beschriebene Top-Down-Ansatz hilft dabei, übersichtliche Programme zu schreiben. Durch fortwährende Unterteilung werden komplizierte Sachverhalte einfach und handhabbar.

- **Variable und Funktionen kosten nichts:** Wenn Sie durch Einführen einer zusätzlichen Variablen oder Funktion die Übersicht erhöhen können, sollten Sie dies tun. Der zunächst offensichtliche Verlust an Speicherplatz oder Rechenzeit wird in den allermeisten Fällen durch die Optimierung des Compilers aufgefangen. Seien Sie daher großzügig mit der Verwendung von Variablen und Funktionen.

- **Code-Verdopplung vermeiden:** Sobald ein Code-Fragment an mehreren Stellen im Programm benötigt wird, sollten Sie dafür eine Funktion schreiben und eventuelle Unterschiede in der Verwendung durch Funktionsparameter kompensieren.

  Das mehrfache Kopieren von Code mit anschließendem Verändern von Details („Cut-and-paste-Programmierung“) ist fehleranfällig: Bei einer nachträglichen Änderung übersieht man leicht, wirklich *alle* Kopien eines Code-Fragments anzupassen. Hierdurch entstehen regelmäßig schwer zu lokalisierende Fehler in Software-Produkten.

- **„Sprechende“ Namen:** Es lohnt sich, Zeit in eine sinnvolle Benennung von Variablen und Funktionen zu investieren. Eine sinnvoll benannte Funktion (z. B. `int is_leap_year (int year)`) dokumentiert das Programms besser als ein Kommentar (z. B. `int a (int b) /*returns 1 if b is a leap year */`). Ein kurzer Name wie z. B. `i` ist für einen lokalen Schleifenindex angebracht, nicht jedoch für eine globale Zustandsvariable oder Bibliotheksfunktion.

- **Globale Variable mit Bedacht einsetzen:** In der Literatur wird oft empfohlen, die Verwendung von globalen Variablen grundsätzlich zu vermeiden und stattdessen Funktionsparameter zu verwenden. Hiervon sollte man nur in begründeten Ausnahmefällen abweichen (z. B. beim Speichern eines globalen Zustands des Programms).

- **Erst konservativ, dann erst ambitioniert programmieren:** In vielen Fällen lohnt es sich, nicht sofort den elegantesten Ansatz zu verfolgen, sondern leichter verständliche Zwischenstufen einzuschieben.

  Beispiel: Die Aufgabe, auf ein gegebenes Datum days Tage zu addieren, läßt sich durch eine Schleife lösen, die days-mal 1 Tag addiert. Dieser Ansatz ist ineffizient, aber narrensicher. Wenn man anschließend einen eleganteren Ansatz ausarbeitet, kann der narrensichere Ansatz beim Testen sehr hilfreich sein.

Bemerkung

Am 3. 1. 2009 meldete *heise online*:

> Kunden des ersten mobilen Media-Players von Microsoft erlebten zum Jahresende eine böse Überraschung: Am 31. Dezember 2008 fielen weltweit alle Zune-Geräte der ersten Generation aus. Ursache war ein interner Fehler bei der Handhabung von Schaltjahren.
>
> http://heise.de/-193332,

Der Artikel verweist auf ein Quelltextfragment, das für einen gegebenen Wert days das Jahr und den Tag innerhalb des Jahres für den days-ten Tag nach dem 1. 1. 1980 berechnen soll:

```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
  {
    if (IsLeapYear (year))
      {
        if (days > 366)
          {
            days -= 366;
            year += 1;
          }
      }
    else
      {
        days -= 365;
        year += 1;
      }
  }
```

Dieses Quelltextfragment weist mehrere Code-Verdopplungen auf:

- Die Anweisung year += 1 taucht an zwei Stellen auf.
- Es gibt zwei unabhängige Abfragen days > 365 und days > 366:
  eine in einer **while**- und die andere in einer **if**-Bedingung.
- Die Länge eines Jahres wird nicht durch eine Funktion berechnet oder in einer Variablen gespeichert; stattdessen werden an mehreren Stellen die expliziten numerischen Konstanten 365 und 366 verwendet.

Diese Probleme führten am 31. Dezember 2008 zu einer Endlosschleife, die sich – z. B. durch eine Funktion DaysInYear() – leicht hätte vermeiden lassen.

Gut hingegen ist die Verwendung einer Präprozessor-Konstanten ORIGINYEAR anstelle der Zahl 1980 sowie die Kapselung der Berechnung der Schaltjahr-Bedingung in einer Funktion IsLeapYear().

4.7.3 Effizienz von Algorithmen

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

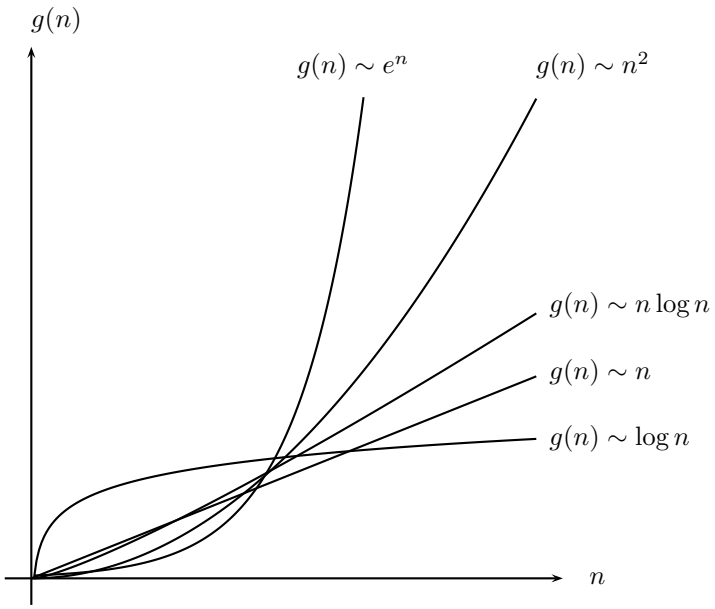
Als Problem wählen wir das Sortieren eines Arrays (z. B. von Namen).

- Minimum/Maximum ermitteln: sort-0.c bis 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.

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 Durschschnitt.

  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

4.8 Verschlüsselung

- Rotation der Buchstaben im Alphabet (Caesar-Chiffre): crypt–1.c
  Spezialfall: um 13 Buchstaben rotieren (ROT13): Verschlüsselung = Entschlüsselung

- Pseudozufallszahlengenerator: crypt-0.c
  rand (): Zufallszahl von 0 bis zu einer großen Zahl RAND_MAX
  rand () % 100: Zufallszahl von 0 bis 99

  srand (137): Pseudozufallszahlengenerator mit der Zahl 137 initialisieren:
  Bei gleichem Startwert erzeugt der Generator dieselbe Folge von Pseudozufallszahlen.

- Verschlüsselung mittels Folge von Pseudozufallszahlen: crypt-1.c
  Jedes Zeichen wird mit einer Pseudozufallszahl Exklusiv-Oder-verknüpft.

  Diese Verschlüsselung ist leicht zu knacken, weil der Pseudozufallszahlengenerator nicht in Hinblick auf Verschlüsselung optimiert ist. Insbesondere gibt es nur 2^32 verschiedene Schlüssel, die ein Computer schnell durchprobieren kann.

- Verschlüsselung mittels einer *speziellen* Folge von Pseudozufallszahlen gemäß dem *RC4*-Algorithmus: crypt-2.c

  Diese Verschlüsselung ist nach aktuellem Stand sicher, wenn mehrere Dinge beachtet werden:

  - Jeder Schlüssel darf nur einmal verwendet werden.
  - Wenn der Anfang des unverschlüsselten Textes oft gleich ist (z. B. Verbindungsaufbau), läßt sich u. U. aus den ersten Bytes des verschlüsselten Textes auf das Passwort schließen. Dies läßt sich vermeiden, indem man die ersten paar Pseudozufallszahlen verwirft. (Aus Gründen der Übersicht wurde in crypt-2.c darauf verzichtet.)

RC4 kommt in der Verschlüsselung von WLAN-Verbindungen zur Anwendung. Die erste Version (WEP) verwendete die Pseudozufallszahlen von Anfang an und war somit unsicher; aktuelle Versionen (WPA) weisen diesen Fehler nicht mehr auf.

5 Dateien

- fhello-1.c ist das klassische „Hello, world!„-Programm in einer Variante, die ausdrücklich die Standardausgabe als Datei anspricht. printf (foo) ist nur eine Abkürzung für fprintf (stdout, foo).
- fhello-2.c schreibt nicht zur Standardausgabe, sondern öffnet dafür eine Datei fhello-2.txt.
- fhello-3.c demonstriert, was passiert, wenn der Versuch, eine Datei zu erstellen, fehlschlägt.
- fhello-4.c zeigt, wie ein Programm sinnvollerweise auf diese Situation reagieren sollte: Es gibt eine Fehlermeldung aus, die den Dateinamen enthält und zusätzliche Informationen, die uns das Betriebssystem als Grund des Fehlschlagens nennt.
- csv.c erzeugt eine Datei mit einer Tabelle im CSV-Format, die mit Tabellenkalkulationsprogrammen weiterverarbeitet werden kann.
- plant-4.c speichert eine rekursive „Pflanze“ in einer Grafikdatei im PBM-Format. PBM ist sehr einfach aufgebaut und ähnelt dem Format, in dem Grafikkarten und Bibliotheken (z. B. OpenGL) Bitmuster im Speicher ablegen.

  Da OpenGL Bilder von unten nach oben aufbaut, PBM jedoch von oben nach unten, steht die Pflanze auf dem Kopf.
- plant-5.c spricht die OpenGL-Zeilen einzeln an und dreht die Pflanze richtigherum.