2.2 Funktionen/Funktionen benutzen

2.2.1 Bausteine für Programme

Wie im vorigen Kapitel beschrieben, zerteilt man bei der Top-Down-Entwurfsmethode das Programmierproblem so lange in kleinere Teilprobleme, bis man zu Teilproblemen kommt, die man einzeln ohne große Schwierigkeit lösen kann. Die einzelnen Teilprobleme werden dann umgesetzt in kleine Programm-Module. Zum Schluss werden die Programm-Module zu einem Gesamtprogramm zusammengefügt.

Fast alle Programmiersprachen haben Möglichkeiten, solche Module (man kann sie auch Bausteine nennen) herzustellen und zusammenzufügen.

2.2.2 Exkurs: Block – Funktion – Modul

In C gibt es drei Ebenen für Bausteine:

a) **Block bzw. Verbundanweisung**
Diese unterste Ebene für Bausteine in C haben wir schon kennengelernt. Es ist der Anweisungsblock, auch Verbundanweisung genannt. Er besteht aus einem Paar geschweifter Klammern, dazwischen können Vereinbarungen und Anweisungen liegen:

```
1  {
2      x=0;                          // default-Wert
3      printf("Eingabe x: ");        // Prompt
4      scanf("%d", &x);              // Eingabe
5      while(getchar()!='\n');       // Puffer leeren
6  }
```

Diese Anweisungen gehören zweifellos zusammen. Deshalb kann man sie in einem Block zusammenfassen. Zu Beginn des Blocks können Vereinbarungen stehen. Das ist aber auch schon alles. Man kann diesen Block nicht mehrfach benutzen. Er steht mitten im Hauptprogramm und behindert die Sicht auf wirklich wichtige Programm-Einzelheiten.

Aus diesem Grund wird der Block fast nur innerhalb von Verzweigungen, Schleifen und Mehrfachauswahlen benutzt.

b) **Funktionsbaustein bzw. Funktion**
Dies ist die mittlere Ebene für Bausteine in C. Bisher haben wir solche Bausteine nur benutzt, aber nicht selbst beschrieben:

```
1  printf("Hello, ");   // benutze printf()-Baustein
2  printf("World!\n"); // benutze printf()-Baustein noch einmal
```

So ein Funktionsbaustein, abgekürzt Funktion genannt, kann im gleichen Programm mehrmals benutzt werden. Und man kann dafür sorgen, dass nicht jedesmal das Gleiche passiert, indem man ihm in der Klammer Daten mitgibt (so genannte Parameter), die seine Arbeit modifizieren.

Diese Art von Baustein wird hier und in den folgenden Kapiteln ausführlich beschrieben. Zuerst reicht es zu wissen, dass ein Funktionsbaustein mehrere Vereinbarungen und Anweisungen und Blöcke enthalten kann.

c) **Modulbaustein bzw. Modul**
Dies ist die obere Ebene für Bausteine in C. Der Modulbaustein kann Programmbausteine enthalten, aber auch eigene Daten speichern, die dem Rest des Programms nicht direkt zur Verfügung stehen.

In C entspricht ein Modulbaustein einer C-Datei. Jeder Modulbaustein kann hier einzeln compiliert werden und damit unabhängig vom Rest des Programms aufgebaut werden.

Diese Art von Baustein wird in einem späteren Kapitel beschrieben.

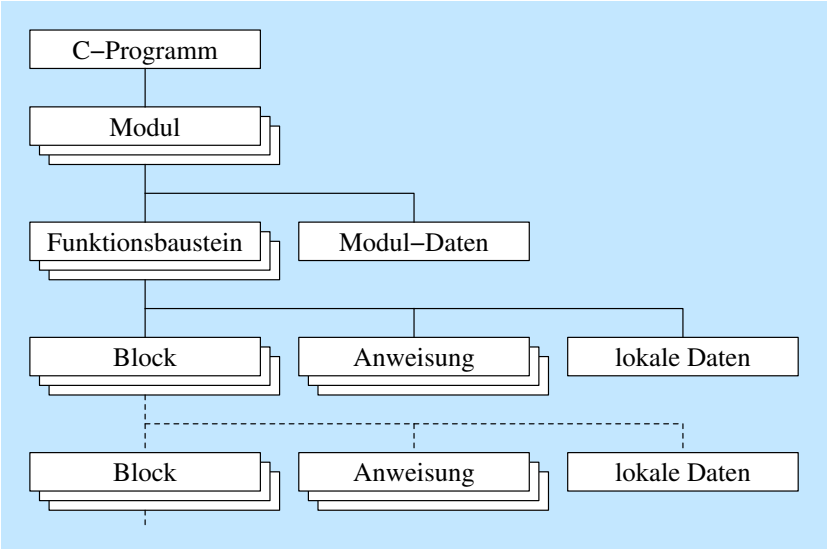

Abbildung 1: Aufbau eines C-Programms aus Modulen, Funktionsbausteinen, Blöcken usw.

Abbildung 1 zeigt, wie ein C-Programm insgesamt aufgebaut ist: Es kann ein oder mehrere Module enthalten. Jedes Modul kann eigene Daten haben und ein oder auch mehrere Funktionsbausteine. Jeder Funktionsbaustein kann sogenannte lokale Daten besitzen und ein oder auch mehrere Blöcke und Anweisungen. Jeder Block kann Daten und Anweisungen besitzen, aber auch weitere Blöcke (gestrichelte Linien). Hier liegt eine Besonderheit: Blöcke kann man verschachteln, d. h., ein Block kann einen oder mehrere weitere Blöcke enthalten. Das haben wir bei Verzweigungen und Schleifen bereits kennengelernt.

Bei Modulen und Funktionsbausteinen ist das anders: Ein Modul entspricht einer C-Datei. Also kann ein Modul keine anderen Module enthalten (sonst müsste eine Datei mehrere andere Dateien enthalten). Genauso ist es mit Funktionsbausteinen: Ein Funktionsbaustein in C kann niemals einen anderen Funktionsbaustein enthalten. Er kann zwar andere Funktionsbausteine benutzen (aufrufen), aber er kann ihn nicht enthalten.

Wie schon angekündigt soll es ab jetzt um die mittlere Ebene für Bausteine in C gehen, nämlich um die Funktionsbausteine.

2.2.3 Aufruf einer Funktion

Hier wird der Funktionsbaustein `printf` benutzt. Man sagt, die Funktion `printf` wird **aufgerufen**:

```
1    printf("Hallo, Welt!\n");
```

Es handelt sich um eine Anweisung[1]. Links steht der Name der Funktion bzw. des Funktionsbausteins. Jede Funktion in C hat einen eindeutigen Namen. Rechts davon steht ein Paar runder Klammern. Dieses Paar runder Klammern sagt aus, dass die Funktion aufgerufen wird. Man braucht keinen Aufruf-Befehl wie `call` oder `machdas` zu schreiben, es reicht das Klammerpaar. In dem Klammerpaar befindet sich die so genannte **Parameterliste**. Mit ihr kann modifiziert werden, was die Funktion macht. Hier soll `printf` den Text `"Hallo, Welt!\n"` ausgeben. Der Text ist dann der **Parameter** (=Modifizierer). Eine Parameterliste kann auch leer sein oder mehr als nur einen Parameter besitzen.

[1] Es ist keine Vereinbarung, weil kein Datentyp vorhanden ist. Es ist auch nicht einfach nur ein Ausdruck, da ein Semikolon am Ende steht.

2.2.4 Beispiele für Funktionen I

Viele Funktionen werden bei einem Standard-C-System bereits mitgeliefert. Ein Standard-C-System besteht nämlich aus:

- dem Compiler und seinen Helfern, die aus unserem C-Quelltext die ausführbare Objektdatei `a.out` machen
- vielen kleinen Funktionen (=Funktionsbausteinen) wie `printf` und `scanf`, die uns eine Menge Arbeit abnehmen können. Die Summe aller bereitgestellten Funktion heißt C-Standard-Bibliothek. Ihr Name wird oft abgekürzt als **libc**.

Während die Sprache C selbst für uns unveränderbar ist, können wir die Funktionen der C-Standard-Bibliothek benutzen, wenn wir wollen. Wir können aber auch die C-Standard-Bibliothek oder einzelne Funktionen daraus durch andere ersetzen, die für unsere Zwecke geeigneter erscheinen.

Jede dieser Funktionen wird in einer Headerdatei beschrieben. So wird `printf` in der Datei `stdio.h` beschrieben. Wenn wir die Funktion benutzen wollen, müssen wir (am besten zu Beginn des Quelltextes) die Datei mit `#include` einbinden.

Die einfachste Funktion der C-Standard-Bibliothek ist die Funktion `abort`. Sie bricht das aktuelle Programm sofort ab (`01abort.c`):

```c
#include <stdio.h>  // fuer printf
#include <stdlib.h> // fuer abort
int main(void)
{
    printf("Hier fange ich an.\n");
    abort();
    printf("Hier hoere ich auf.\n");
    return 0;
}
```

Terminal

```
schueler@debian964:~$ gcc 01abort.c
schueler@debian964:~$ a.out
Hier fange ich an.
Abgebrochen
```

Diese Funktion wird benutzt, wenn absolut keine weitere Aktion mehr ausgeführt werden soll. Sie hat keinen Parameter, das Paar runder Klammern ist leer. Man darf es aber nicht weglassen, weil das Programm die Anweisung sonst anders versteht[2].

In manchen Varianten der C-Standard-Bibliothek findet man die Funktion `sleep`, die nicht zum C-Standard gehört. Wenn man `sleep` benutzt, ist das Programm nicht mehr portabel, d. h., es kann nicht auf allen C-Systemen compiliert werden, und es kann sein, dass es auf einem anderen C-System anders funktioniert. Bei der Linux-Variante bewirkt der Aufruf `sleep(3)`, dass das Programm drei Sekunden wartet (`02sleep.c`):

```c
#include <stdio.h>  // fuer printf
#include <unistd.h> // fuer sleep, nicht Standard-C!
int main(void)
{
    printf("Hier fange ich an.\n");
    sleep(3);
    printf("Hier hoere ich auf.\n");
```

[2] Es würde dann die Startadresse der Funktion holen. Siehe im Kapitel über Funktionszeiger.

```
8     return 0;
9 }
```

Hier besteht die Parameterliste aus einer ganzen Zahl.

Im folgenden Beispiel haben wir eine Parameterliste aus zwei Parametern (`03setlocale.c`):

```
1 #include <stdio.h>  // fuer printf
2 #include <locale.h> // fuer setlocale und LC_ALL
3 int main(void)
4 {
5     printf("Hier fange ich an mit x=%g.\n", 3.5);
6     setlocale(LC_ALL, "");
7     printf("Hier hoere ich auf mit x=%g.\n", 3.5);
8     return 0;
9 }
```

Die zwei Parameter werden durch ein Komma getrennt, typisch für eine Liste in einer Programmiersprache. Der Parameter `LC_ALL` ist eine symbolische Konstante, die einer bestimmten Zahl entspricht.

Es gibt außerdem Funktionsbausteine, die drei oder mehr Parameter erwarten. Meistens sind die Anzahl und die Art der Parameter festgelegt: `abort` hat keinen, `sleep` einen und `setlocale` zwei Parameter. Es gibt seltene Ausnahmen von dieser Regel. `printf` und `scanf` sind solche Ausnahmen:

```
1     printf("Hallo\n");                    // 1 Parameter
2     printf("x=%i\n", x);                  // 2 Parameter
3     printf("x=%i , y=%i\n", x, y);        // 3 Parameter
4     printf("x=%i , y=%i , z=%i\n", x, y, z); // 4 Parameter
```

2.2.5 Was passiert bei einem Funktionsaufruf?

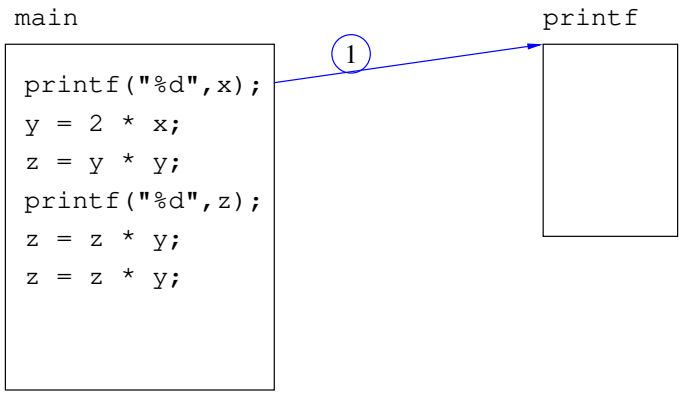

Abbildung 2: Aufruf einer Funktion, Teil 1

Beim Aufruf einer Funktion springt der Programmzähler der CPU zum Anfang des Codes des Funktionsbausteins (Abbildung 2). Das ist aber noch nicht alles. Am Ende des Funktionsbausteins erfolgt wieder ein Sprung, nämlich der Rücksprung zum Hauptprogramm (Abbildung 3, Pfeil Nr. 2).

Man kann in einem Programm auch zweimal `printf` aufrufen. Beide Male erfolgt ein Sprung zur gleichen Adresse (Abbildung 4, Pfeil Nr. 3).

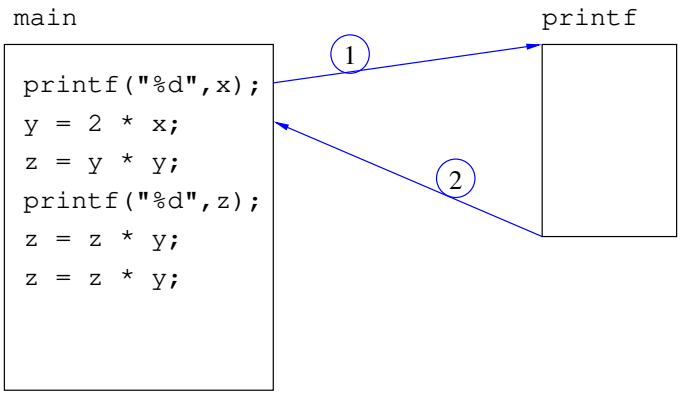

Abbildung 3: Aufruf einer Funktion, Teil 2

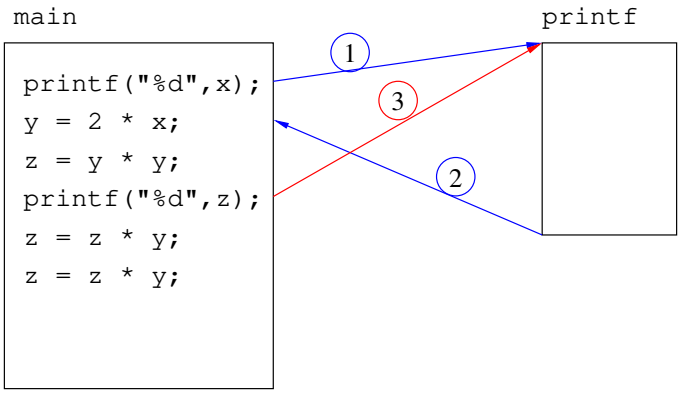

Abbildung 4: Zweiter Aufruf einer Funktion, Teil 1

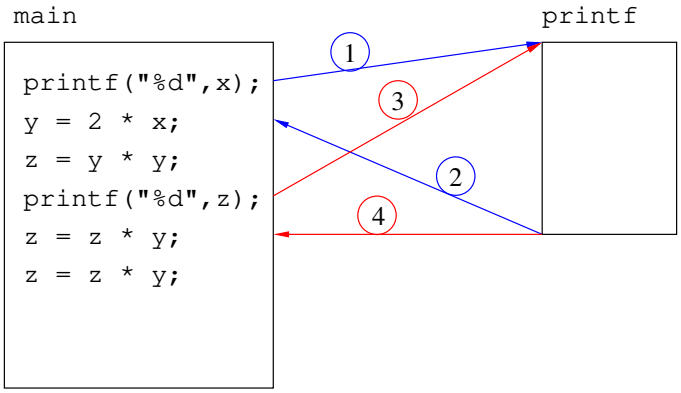

Abbildung 5: Zweiter Aufruf einer Funktion, Teil 2

Aber: In den beiden Fällen erfolgt der Rücksprung immer hinter die Adresse, von wo aus gesprungen wurde (Abbildung 5, Pfeil Nr. 4). Offenbar hat sich jemand die sogenannte **Rücksprungadresse** gemerkt!

Damit das funktioniert, ist ein Funktionsaufruf etwas anderes als ein einfacher Sprung. Sowohl in C als auch für die CPU[3].

Der Funktionsaufruf (ebenso der CPU-Befehl call) besteht also in Wirklichkeit aus zwei Teilen[4]:

a) aktuelle Adresse sichern (meistens auf dem so genannten Stack)

b) zur Startadresse von `printf` springen

Genauso besteht der Rücksprung am Ende der Funktion (ebenso wie der CPU-Befehl ret) auch aus zwei Teilen:

a) gesicherte Adresse zurückholen

b) an diese Adresse springen

Durch diesen Kunstgriff kann man eine Funktion an beliebigen Stellen des Programms immer wieder benutzen.

2.2.6 Beispiele für Funktionen II

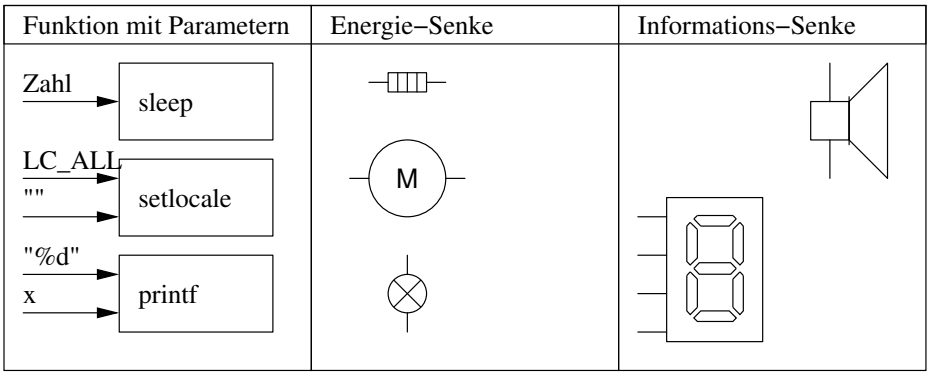

Abbildung 6: Funktionen mit Parametern und ihre Entsprechung in der Elektro- und Informationstechnik

Bis hierher ging es um Funktionen, die für sich stehen (z. B. `abort`) oder Parameter haben. Allgemein kann man sagen, dass Parameter die Eingabedaten einer Funktion sind. Insofern kann man die hier gezeigten Funktionen vergleichen mit Energie- oder Informationssenken in der Elektrotechnik: Es fließt nur Information oder Energie in den Baustein hinein (Abbildung 6).

Ab jetzt sollen Funktionen vorgestellt werden, die Daten ausgeben. Diese Funktionen entsprechen Energie- oder Informations-Quellen in der Elektrotechnik. Es fließt Information oder Energie aus dem Baustein heraus (Abbildung 7).

Ein Beispiel ist die Funktion `rand`, die in `stdlib.h` beschrieben ist (`04rand.c`):

```
1 #include <stdio.h>  // fuer printf
2 #include <stdlib.h> // fuer rand
3 int main(void)
4 {
5     int x;
```

[3]C hat auch eine Sprunganweisung im Angebot. Sie heißt goto. Auf der CPU-Ebene heißt der Befehl (je nach CPU) meistens `jump` oder `jmp`.

[4]Außerdem müssen noch Parameter übergeben werden, falls vorhanden. Dazu später.

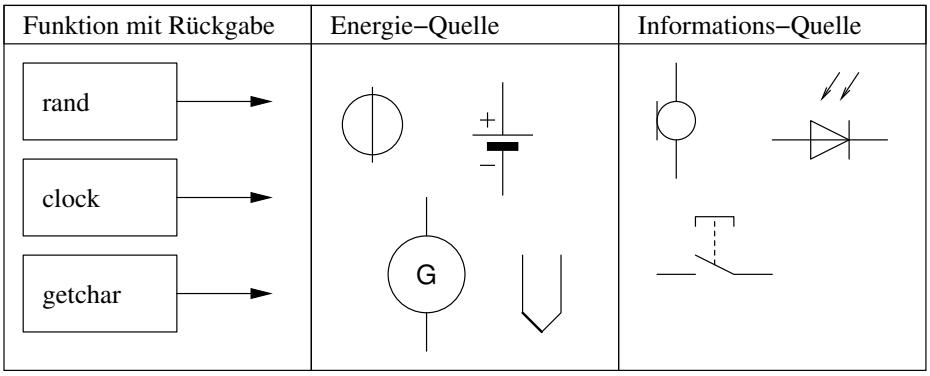

Abbildung 7: Funktionen mit Rückgabe und ihre Entsprechung in der Elektro- und Informationstechnik

```
6     x=rand();
7     printf("Zufallszahl: %i\n", x);
8     x=rand();
9     printf("Zufallszahl: %i\n", x);
10    return 0;
11 }
```

In Zeile 6 wird die Funktion `rand` aufgerufen, erkennbar am Klammerpaar. Wenn sie fertig ist, hat sie einen Rückgabewert. Dieser Wert wird dort ingesetzt, wo der Funktionsaufruf stand. Falls der Rückgabewert 30 war, steht dort also quasi: `x=30;`

Der Rückgabewert einer Funktion ist vergleichbar mit dem Wert eines Ausdrucks. Bei der Zeile `x=(20+50);` wird zuerst der Wert der Summe berechnet. Wenn die Berechnung fertig ist, wird der Wert des Ausdrucks (also das Ergebnis) an der Stelle eingesetzt, an der vorher die Klammer mit der Rechenaufgabe stand: `x=70;`.

Die Funktion `rand` ist ein Zufallsgenerator. Wenn man im gleichen Programm mehrmals nacheinander `rand` aufruft, wird immer wieder ein neuer Zufallswert berechnet.

Weitere Beispiele sind die Funktionen `clock`, beschrieben in `time.h`, und `getchar`, beschrieben in `stdio.h` (`05clock+getchar.c`):

```
1  #include <stdio.h> // fuer printf und getchar
2  #include <time.h>  // fuer clock
3  int main(void)
4  {
5     int x;
6     x = clock();
7     printf("Laufzeit: %i ticks\n", x);
8     printf("Zeichen eingeben und Enter druecken:");
9     x = getchar();
10    while(getchar()!='\n'){}
11    printf("ASCII–Zeichen: %i\n", x);
12    x = clock();
13    printf("Laufzeit: %i ticks\n", x);
14    return 0;
15 }
```

Die Funktion `clock` ermittelt die Laufzeit des Programmes und gibt sie zurück; die Funktion `texttt` nimmt ein Zeichen von der Tastatur entgegen und gibt die entsprechende Zahl zurück (siehe ASCII-Tabelle).

2.2.7 Beispiele für Funktionen III

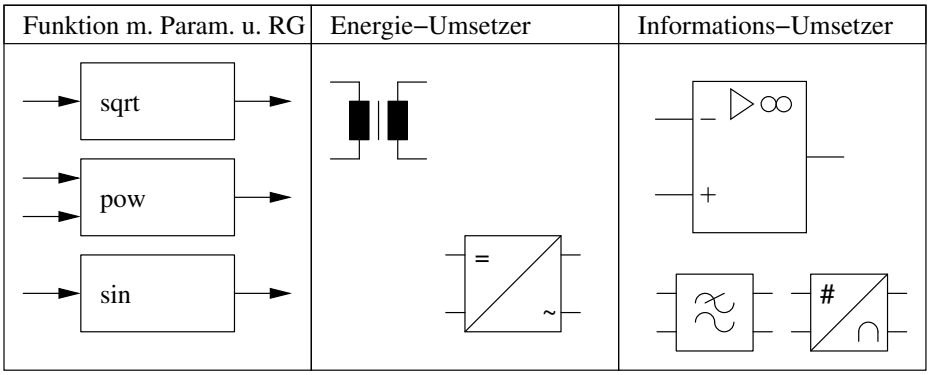

Abbildung 8: Funktionen mit Parametern und Rückgabe und ihre Entsprechung in der Elektro- und Informationstechnik

Schließlich gibt es auch Funktionen, die Daten annehmen und andere Daten ausgeben. In der Elektrotechnik entspricht dies Baugruppen, die Energie oder Information umsetzen, verstärken oder filtern (Abbildung 8).

Hier sollen beispielhaft die Funktionen `sqrt` und `pow` vorgestellt werden, die beide in `math.h` beschrieben werden (`06sqrt+pow.c`):

```c
#include <stdio.h> // fuer printf
#include <math.h>  // fuer sqrt und pow
int main(void)
{
    double x, y, z;
    x = 49.0;
    y = sqrt(x);
    printf("Wurzel von 49: %g\n", y);
    x = 10.0;
    y = 3.0;
    z = pow(x, y); // z <- x hoch y
    printf("10 hoch 3: %g\n", z);
    return 0;
}
```

Der Ausdruck `sqrt(49.0)` berechnet $\sqrt{49}$, der Ausdruck `pow(10.0,3.0)` berechnet 10^3.

Um dieses Programm zu compilieren, muss man bei manchen Systemen dem GCC eine zusätzliche Option auf der Befehlszeile mitgeben:

Terminal

```
schueler@debian964:~$ gcc 06sqrt+pow.c
/usr/bin/ld: /tmp/cclKyBnJ.o: in function `main':
06sqrt+pow.c:(.text+0x1b): undefined reference to `sqrt'
/usr/bin/ld: 06sqrt+pow.c:(.text+0x63): undefined reference to `pow'
collect2: error: ld returned 1 exit status
schueler@debian964:~$ gcc 06sqrt+pow.c -lm
schueler@debian964:~$ a.out
Wurzel von 49: 7
10 hoch 3: 1000
```

Die Option `-lm` bindet alle die Funktionen für Gleitkommazahlen ein, die in `math.h` beschrieben werden.