

Ein Buch zum Mitmachen und Verstehen

Mit  
Arduino-  
Workshop

# C

## von Kopf bis Fuß



Entdecken Sie die  
Geheimnisse der C-Gurus



Vermeiden  
Sie peinliche  
Zeigerfehler

Spielen Sie mit  
der C-Standard-  
bibliothek herum



Finden Sie  
heraus, wie  
make Ihr Leben  
verändern kann

Erfahren Sie,  
wie variadische  
Funktionen Susi zu  
mehr Flexibilität  
verhelfen



Reanimieren Sie  
ein klassisches  
Arcade-Spiel



O'REILLY®

David Griffiths / Dawn Griffiths, C von Kopf bis Fuß, O'Reilly, ISBN 978-3-86899-386-8

David Griffiths & Dawn Griffiths  
Deutsche Übersetzung von Lars Schulten

# Der Inhalt (im Überblick)

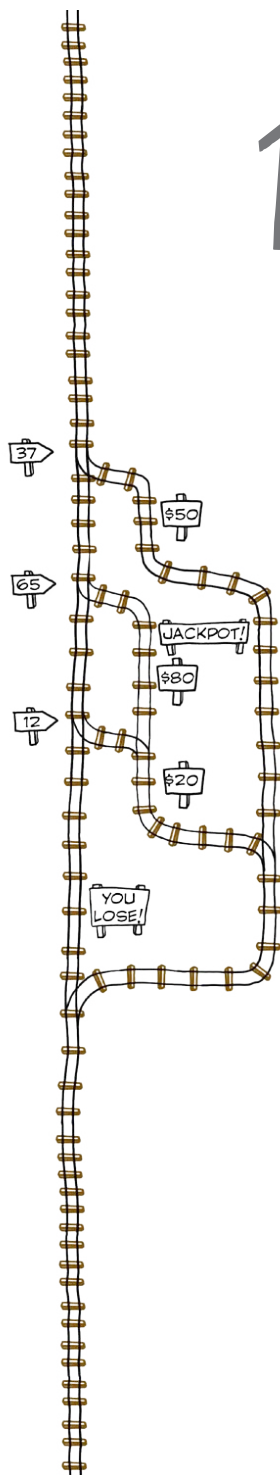
|     |                                                                            |     |
|-----|----------------------------------------------------------------------------|-----|
|     | Einführung                                                                 | xxv |
| 1   | Erste Schritte mit C: <i>Eintauchen</i>                                    | 1   |
| 2   | Speicher und Zeiger: <i>Worauf zeigst du?</i>                              | 41  |
| 2.5 | Strings: <i>Stringtheorie</i>                                              | 83  |
| 3   | Kleine Werkzeuge erstellen: <i>Eine Sache tun, und das gut</i>             | 103 |
| 4   | Mehrere Quelldateien: <i>Zerlegen und zusammenbauen</i>                    | 157 |
|     | 1. C-Projekt: <i>Arduino</i>                                               | 207 |
| 5   | Structs, Unions und Bitfelder: <i>Eigene Strukturen</i>                    | 217 |
| 6   | Datenstrukturen und dynamischer Speicher: <i>Brücken bauen</i>             | 267 |
| 7   | Fortgeschrittene Funktionen: <i>Ihre Funktionen auf Vordermann bringen</i> | 311 |
| 8   | Statische und dynamische Bibliotheken: <i>Code-Wiederverwendung</i>        | 351 |
|     | 2. C-Projekt: <i>OpenCV</i>                                                | 389 |
| 9   | Prozesse und Systemaufrufe: <i>Grenzverletzungen</i>                       | 397 |
| 10  | Interprozesskommunikation: <i>Ein nettes Gespräch</i>                      | 429 |
| 11  | Sockets und Netzwerke: <i>127.0.0.1 ist ein toller Ort</i>                 | 467 |
| 12  | Threads: <i>Parallelwelten</i>                                             | 501 |
|     | 3. C-Projekt: <i>Blasteroids</i>                                           | 523 |
| A   | Was übrig bleibt: <i>Die Top Ten (nicht behandelt)</i>                     | 539 |
| B   | C-Themen: <i>Zusammenfassungen im Überblick</i>                            | 553 |

# Der Inhalt (jetzt ausführlich)

## Einführung

**Ihr Gehirn und C.** Sie versuchen, etwas zu *lernen*, und Ihr *Hirn* tut sein Bestes, damit das Gelernte nicht *hängen bleibt*. Es denkt nämlich: »Wir sollten lieber ordentlich Platz für wichtigere Dinge lassen, z. B. für das Wissen darüber, welche Tiere einem gefährlich werden könnten oder dass es eine ganz schlechte Idee ist, nackt Snowboard zu fahren.« Tja, wie schaffen wir es nun, Ihr Gehirn davon zu überzeugen, dass Ihr Leben davon abhängt, etwas über C zu wissen?

|                                        |       |
|----------------------------------------|-------|
| Für wen ist dieses Buch?               | xxvi  |
| Wir wissen, was Sie denken.            | xxvii |
| Metakognition                          | xxix  |
| So machen Sie sich Ihr Gehirn untertan | xxxi  |
| Lies mich                              | xxxii |
| Die technischen Gutachter              | xxxiv |
| Danksagungen                           | xxxv  |



# Erste Schritte mit C

## Eintauchen

### Wollen Sie dem Rechner auf die Finger schauen?

Müssen Sie **Hochleistungscode** für ein neues Spiel schreiben? Einen **Arduino**-Controller programmieren? Oder diese raffinierte **externe Bibliothek** in Ihrer iPhone-App einsetzen? Wenn das der Fall ist, wird C Ihr bester Freund werden. C arbeitet auf einer **viel elementarerer Ebene** als die meisten anderen Programmiersprachen. Wenn Sie C verstanden haben, werden Sie auch viel besser verstehen, **was eigentlich im Herzen der Maschine vor sich geht**. C kann Ihnen sogar helfen, andere Sprachen besser zu verstehen. Zögern Sie nicht! Machen Sie Ihren Compiler bereit – auf dass Sie schon bald loslegen können.

|                                                  |    |
|--------------------------------------------------|----|
| C, die Sprache für kleine, schnelle Programme    | 2  |
| Aber wie sieht ein vollständiges C-Programm aus? | 5  |
| Aber wie führen Sie das Programm aus?            | 9  |
| Zwei Arten von Befehlen                          | 14 |
| So sieht der Code bislang aus                    | 15 |
| Kartenzählen? In C?                              | 17 |
| Vergleiche kennen nicht nur Gleichheit ...       | 18 |
| Wie sieht der Code jetzt aus?                    | 25 |
| Weichen stellen                                  | 26 |
| Wenn einmal keinmal ist ...                      | 29 |
| Schleifen haben oft eine ähnliche Struktur ...   | 30 |
| Mit break ausbrechen ...                         | 31 |
| Ihr C-Werkzeugkasten                             | 40 |

# Speicher und Zeiger

## Worauf zeigst du?

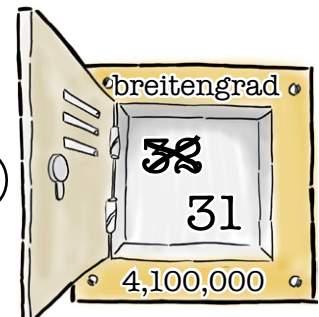
2

Wenn Sie C wirklich beherrschen wollen, müssen Sie verstehen, wie C mit Speicher umgeht.

C bietet Ihnen ziemlich umfangreiche Möglichkeiten, zu *steuern*, wie Ihr Programm den **Speicher des Systems** nutzt. In diesem Kapitel werden wir den Vorhang lüften und Ihnen zeigen, was passiert, wenn Sie **Variablen lesen und schreiben**. Sie werden erfahren, **wie Arrays funktionieren**, wie man einige **garstige Speicherprobleme vermeidet**, und natürlich auch einsehen lernen, dass der Weg zum gewieften C-Programmierer nur über **die Beherrschung von Zeigern und der Speicheradressierung** führt.



|                                                        |    |
|--------------------------------------------------------|----|
| C-Code enthält Zeiger                                  | 42 |
| Ein Blick in den Speicher                              | 43 |
| Segel setzen mit Zeigern                               | 44 |
| Versuchen wir, der Variablen einen Zeiger zu übergeben | 47 |
| Speicherzeiger einsetzen                               | 48 |
| Wie übergibt man einer Funktion einen String?          | 53 |
| Array-Variablen sind wie Zeiger ...                    | 54 |
| Was der Computer denkt, wenn er Ihren Code ausführt    | 55 |
| Aber Array-Variablen sind nicht das Gleiche wie Zeiger | 59 |
| Warum Arrays wirklich mit 0 beginnen                   | 61 |
| Warum Zeiger Typen haben                               | 62 |
| Zeiger für die Dateneingabe verwenden                  | 65 |
| Aufgepasst mit scanf()                                 | 66 |
| fgets() ist eine Alternative zu scanf()                | 67 |
| Stringlitterale können nie aktualisiert werden         | 72 |
| Wenn Sie einen String ändern wollen, kopieren Sie ihn  | 74 |
| Speicher speichern                                     | 80 |
| Ihr C-Werkzeugkasten                                   | 81 |



# 2.5

## Strings

### Stringtheorie

#### Strings muss man nicht nur lesen.

Sie haben erfahren, dass Strings in C eigentlich `char-Arrays` sind, aber was C Sie mit ihnen *anstellen* lässt, das wissen Sie noch nicht. Dazu müssen wir uns **`string.h`** zuwenden. `string.h` ist ein Teil der C-Standardbibliothek, der sich ganz der **Stringmanipulation** widmet. Wenn Sie Strings **verketteten**, einen String in einen anderen **kopieren** oder zwei Strings **vergleichen** wollen, können die Funktionen in `string.h` nützlich sein. In diesem Kapitel werden Sie sehen, wie Sie ein **Array mit Strings** erstellen, bevor wir uns genauer ansehen werden, wie man mit der Funktion `strstr()` **in Strings sucht**.

|                                            |     |
|--------------------------------------------|-----|
| Frank verzweifelt gesucht                  | 84  |
| Ein Array mit Arrays erstellen             | 85  |
| Strings finden, die den Suchtext enthalten | 86  |
| Die Funktion <code>strstr()</code> nutzen  | 89  |
| Zeit, dass wir uns unseren Code ansehen    | 94  |
| Array von Arrays vs. Array von Zeigern     | 98  |
| Ihr C-Werkzeugkasten                       | 101 |



## Kleine Werkzeuge erstellen

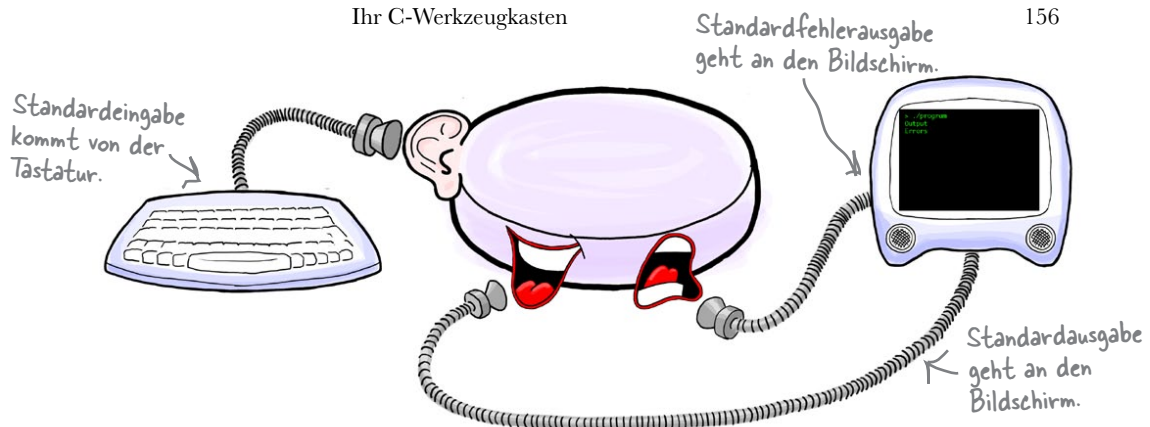
# 3

### Eine Sache tun, und das gut

#### Alle Betriebssysteme beinhalten kleine Werkzeuge.

Kleine in C geschriebene Werkzeuge erledigen **kleine spezielle Aufgaben**, schreiben oder lesen Dateien oder filtern Daten. Wenn Sie komplexere Aufgaben bewältigen müssen, können Sie *mehrere Werkzeuge hintereinanderschalten*. Aber wie werden diese kleinen Werkzeuge erstellt? In diesem Kapitel werden wir uns die Bausteine der Erstellung kleiner Werkzeuge anschauen. Sie werden lernen, wie man **Kommandozeilenoptionen** steuert, wie man **Informationsströme** verarbeitet und **Umleitungen** einsetzt, um im Handumdrehen Werkzeuge zu bauen.

|                                                                         |     |
|-------------------------------------------------------------------------|-----|
| Kleine Werkzeuge können große Probleme lösen                            | 104 |
| So sollte das Programm funktionieren                                    | 108 |
| Aber Sie nutzen noch keine Dateien ...                                  | 109 |
| Sie können Ihre Daten umleiten                                          | 110 |
| Die Standardfehlerausgabe                                               | 120 |
| Standardmäßig wird die Standardfehlerausgabe an den Bildschirm gebunden | 121 |
| fprintf() schreibt in einen Datenstrom                                  | 122 |
| Aktualisieren wir den Code, damit er fprintf() nutzt                    | 123 |
| Kleine Werkzeuge sind flexibel                                          | 128 |
| Ändern Sie geo2json nicht                                               | 129 |
| Eine andere Aufgabe erfordert ein anderes Werkzeug                      | 130 |
| Eingaben und Ausgaben mit einer Pipe verbinden                          | 131 |
| Das bermuda-Werkzeug                                                    | 132 |
| Aber was ist, wenn Sie mehr als eine Datei ausgeben wollen?             | 137 |
| Der eigene Datenstrom                                                   | 138 |
| main() kann mehr                                                        | 141 |
| Lassen Sie die Bibliothek für sich wirken                               | 149 |
| Ihr C-Werkzeugkasten                                                    | 156 |





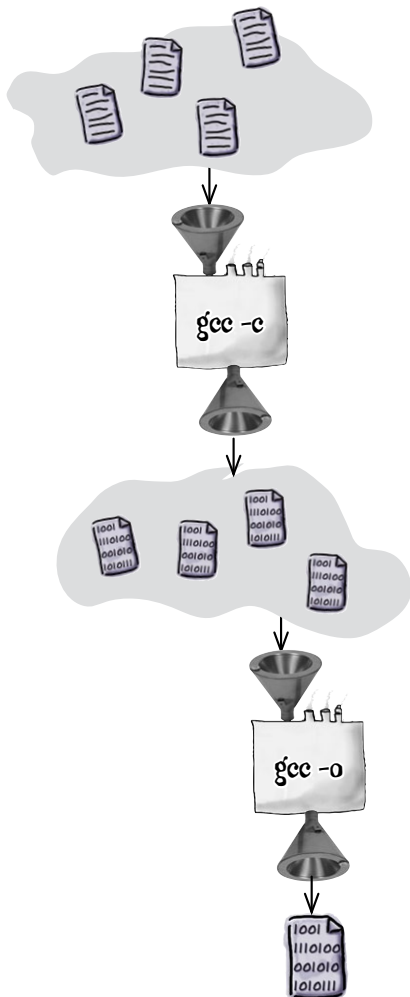
## Mehrere Quelldateien

### Zerlegen und zusammenbauen

# 4

**Wenn Sie ein großes Programm erstellen, heißt das nicht, dass Sie auch eine große Quelldatei haben wollen.**

Können Sie sich vorstellen, wie schwierig und zeitaufwendig die Wartung einer einzigen Quelldatei bei umfangreichen Programmen werden kann? In diesem Kapitel werden Sie erfahren, wie Ihnen C ermöglicht, Quellcode in **kleine, handhabbare Happen** zu zerlegen und diese dann zu **einem großen Programm** zusammenzusetzen. Auf dem Weg dorthin werden Sie etwas mehr über die **Feinheiten von Datentypen** erfahren und werden jemandem über den Weg laufen, der einer Ihrer besten Freunde werden wird: **make**.



|                                                     |     |
|-----------------------------------------------------|-----|
| Datentypen-Schnellkurs                              | 162 |
| Großes darf man nicht in Kleines stecken            | 163 |
| Mit Casts floats in ganze Zahlen packen             | 164 |
| Oh nein, arbeitslose Schauspieler am Werk ...       | 168 |
| Schauen wir uns an, was dem Code widerfahren ist    | 169 |
| Compiler mögen keine Überraschungen                 | 171 |
| Die Deklaration von der Definition trennen          | 173 |
| Ihre erste Header-Datei                             | 174 |
| Bei Gemeinsamkeiten ...                             | 182 |
| Sie können Code auf mehrere Dateien aufteilen       | 183 |
| Kompilieren – was steckt dahinter?                  | 184 |
| Der gemeinsame Code braucht einen Header            | 186 |
| Es ist kein Mysterium ... oder doch?                | 189 |
| Nicht alles neu kompilieren                         | 190 |
| Erst aus Quellen Objektdateien machen               | 191 |
| Das Nachhalten der Dateien ist aufwendig            | 196 |
| Die Erstellung mit make automatisieren              | 198 |
| Wie make funktioniert                               | 199 |
| make mit einem makefile über Ihren Code informieren | 200 |
| Abheben!                                            | 205 |
| Ihr C-Werkzeugkasten                                | 206 |

# 1. C-Workshop

## Arduino

Hatten Sie sich schon einmal gewünscht, dass Ihre Pflanzen Ihnen sagen könnten, wann sie Wasser brauchen? Mit einem Arduino geht das! In diesem Workshop werden Sie eine Arduino-gesteuerte und in C programmierte Pflanzenüberwachung konstruieren.





# Structs, Unions und Bitfelder

## 5

### Eigene Strukturen

#### Die meisten Dinge im Leben sind komplexer als einfache Zahlen.

Bislang haben wir uns die elementaren Datentypen der Programmiersprache C angesehen, aber was ist, wenn Zahlen und einfache Zeichenfolgen nicht mehr ausreichen? Was ist, wenn Sie **Dinge aus dem wahren Leben modellieren wollen**? **structs** ermöglichen Ihnen, die **Komplexität der realen Welt zu modellieren**, indem Sie eigene Strukturen gestalten. In diesem Kapitel werden Sie lernen, wie Sie die **elementaren Datentypen** zu Strukturen **kombinieren** und mit Unions die **Unwägbarkeiten des Lebens in den Griff bekommen**. Und wenn Ihnen ein einfaches Ja oder Nein ausreicht, können **Bitfelder** genau das Richtige für Sie sein.

|                                                        |     |
|--------------------------------------------------------|-----|
| Wenn viele Daten wandern wollen                        | 218 |
| Bürogespräche                                          | 219 |
| Strukturierte Datentypen mit Structs gestalten         | 220 |
| Fisch und nichts anderes                               | 221 |
| Die Felder eines Struct lesen Sie mit dem ».«-Operator | 222 |
| Kann man ein struct in ein anderes stecken?            | 227 |
| Wie aktualisiert man ein struct?                       | 236 |
| Der Code klonet die Kröte                              | 238 |
| Sie brauchen einen Zeiger auf das struct               | 239 |
| (*)s.alter vs. *s.alter                                | 240 |
| Manchmal erfordert eine Sache mehrere Datentypen       | 246 |
| Mit einer Union können Sie Speicherplatz sparen        | 247 |
| Wie man eine Union nutzt?                              | 248 |
| Eine Enum-Variable speichert ein Symbol                | 255 |
| Kontrolle bis zur Bit-Ebene                            | 261 |
| Bitfelder speichern eine beliebige Anzahl von Bits     | 262 |
| Ihr C-Werkzeugkasten                                   | 266 |

Das ist Myrtle ...



... aber die Funktion erhält Ihren Klon.



Turtle "t"

xiv



# Datenstrukturen und dynamischer Speicher

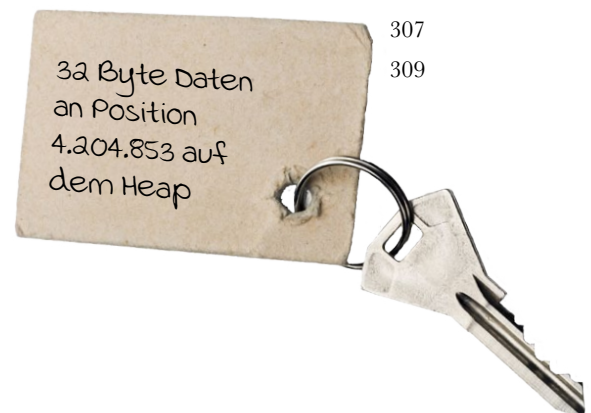
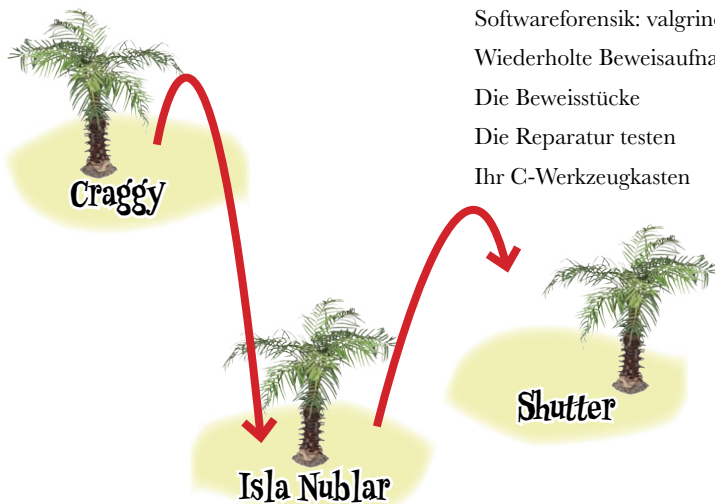
## Brücken bauen

# 6

### Manchmal reicht ein Struct einfach nicht aus.

Zur Modellierung komplexer Datenanforderungen muss man häufig Structs **verbinden**. In diesem Kapitel werden Sie erfahren, wie Sie Struct-**Zeiger** nutzen können, um eigene Datentypen zu **großen, komplexen Datenstrukturen** zusammenzuknüpfen. Sie werden *Schlüsselprinzipien* erforschen, indem Sie **verkettete Listen** erstellen. Sie werden auch erfahren, wie Sie Ihre Datenstrukturen dazu bringen, flexible Datenmengen zu bewältigen, indem Sie **dynamisch Speicher auf dem Heap allozieren** und wieder freigeben, wenn Sie ihn nicht mehr benötigen. Und dann werden wir Ihnen ein Werkzeug vorstellen, das Sie unterstützen kann, wenn Sie Probleme bei der Haushaltsführung haben: **valgrind**.

|                                                            |     |
|------------------------------------------------------------|-----|
| Flexibler Speicher gefällig?                               | 268 |
| Verkettete Listen sind wie Datenketten                     | 269 |
| Verkettete Listen gestatten Einfügungen                    | 270 |
| Eine rekursive Struktur erstellen                          | 271 |
| Mit C Inseln schaffen ...                                  | 272 |
| Werte in die Liste einsetzen                               | 273 |
| Dynamische Speicherung auf dem Heap                        | 278 |
| Hinterher den Speicher freigeben                           | 279 |
| Mit malloc() Speicher anfordern ...                        | 280 |
| Reparieren wir den Code mit strdup()                       | 286 |
| Geben Sie Speicher frei, wenn Sie ihn nicht mehr benötigen | 290 |
| Das System im Überblick                                    | 300 |
| Softwareforensik: valgrind                                 | 302 |
| Wiederholte Beweisaufnahme mit valgrind                    | 303 |
| Die Beweisstücke                                           | 304 |
| Die Reparatur testen                                       | 307 |
| Ihr C-Werkzeugkasten                                       | 309 |



# Fortgeschrittene Funktionen

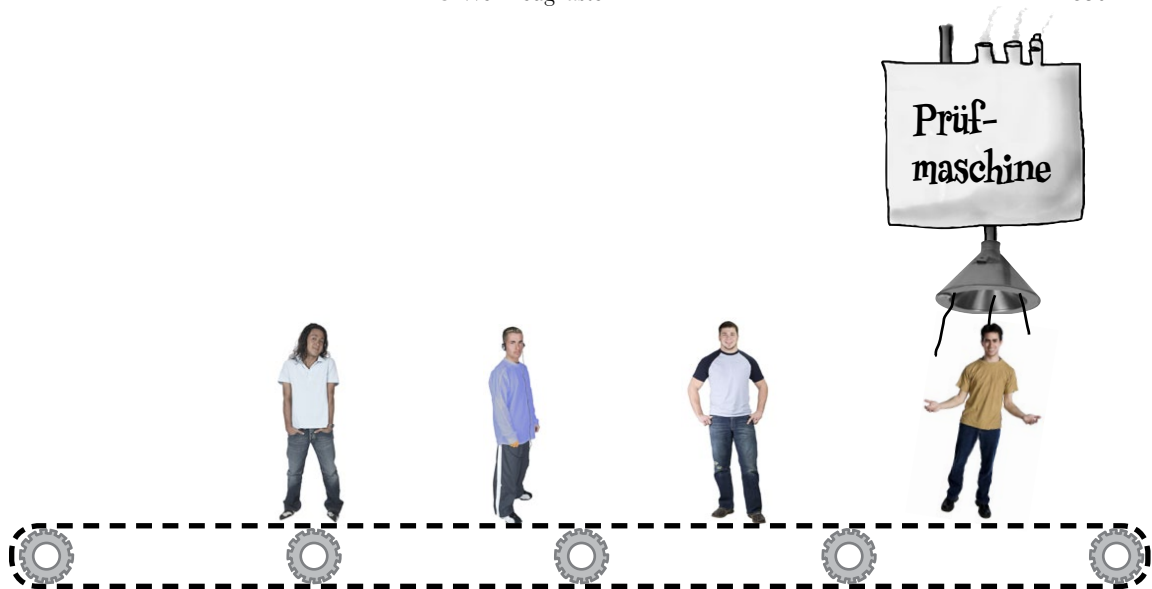
## 7

### Ihre Funktionen auf Vordermann bringen

Einfache Funktionen sind schön, aber reichen manchmal nicht aus.

Bislang haben Sie sich auf die Grundlagen beschränkt, aber was ist, wenn Sie mehr *Macht* und *Flexibilität* benötigen, um Ihr Ziel zu erreichen? In diesem Kapitel werden Sie erfahren, wie Sie **den IQ Ihres Codes aufpeppen können**, indem Sie **Funktionen als Parameter übergeben**. Sie werden erfahren, wie man **Dinge mit Vergleichsfunktionen sortiert**. Und schließlich werden Sie entdecken, wie Sie Ihren Code mit **variadischen Funktionen anpassungsfähiger** machen.

|                                                         |     |
|---------------------------------------------------------|-----|
| Auf der Suche nach dem Märchenprinzen ...               | 312 |
| Einer Funktion Code übergeben                           | 316 |
| Sie müssen suchen() den Namen einer Funktion mitteilen  | 317 |
| Jeder Funktionsname ist ein Zeiger auf die Funktion ... | 318 |
| ... aber es gibt keinen Funktionsdatentyp               | 319 |
| Wie man Funktionszeiger erstellt                        | 320 |
| Ordnung mit der C-Standardbibliothek                    | 325 |
| Ordnung schaffen mit Funktionszeigern                   | 326 |
| Standardbriefe automatisieren                           | 334 |
| Ein Array von Funktionszeigern erstellen                | 338 |
| Funktionen dehnbar machen                               | 343 |
| Ihr C-Werkzeugkasten                                    | 350 |



# Statische und dynamische Bibliotheken

## 8

### Code-Wiederverwendung

#### Die Macht der Standardbibliothek haben Sie bereits kennengelernt.

Jetzt ist es an der Zeit, dass Sie diese Macht für Ihren *eigenen* Code einsetzen. In diesem Kapitel werden Sie lernen, wie Sie Ihre **eigenen Bibliotheken** erstellen und den **gleichen Code in mehreren Programmen wiederverwenden**. Außerdem werden Sie erfahren, wie Ihnen **dynamische Bibliotheken** ermöglichen, Ihren Code zur Laufzeit zu teilen. Sie werden in die Geheimnisse der *Programmierungsgurus* eingeweiht werden. Und wenn Sie das Ende dieses Kapitels erreicht haben, werden Sie Code schreiben können, der gut skaliert und leicht und effizient zu warten ist.

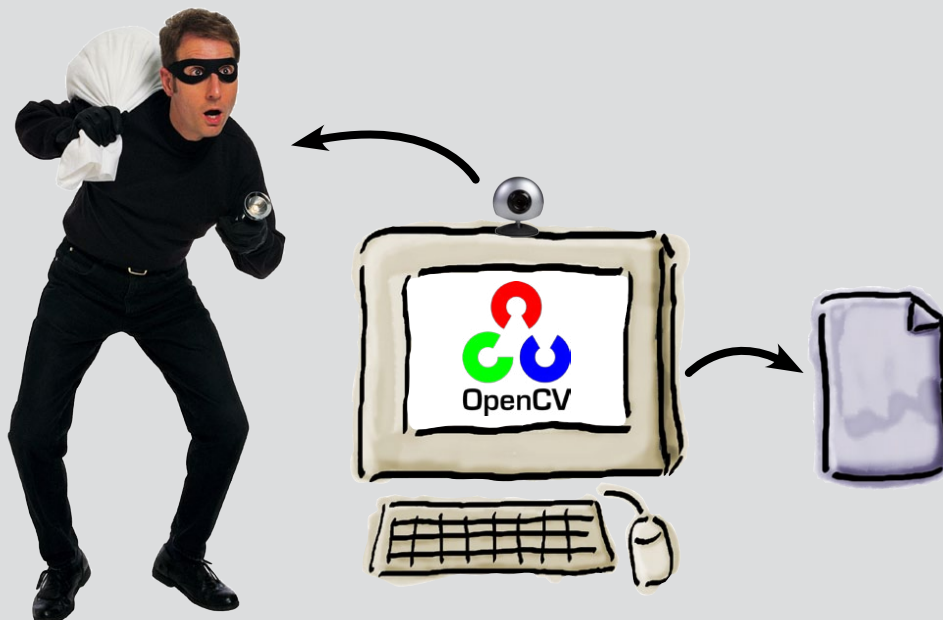
|                                                             |     |
|-------------------------------------------------------------|-----|
| Code, den Sie auf die Bank bringen können                   | 352 |
| Spitze Klammern sind für Standard-Header                    | 354 |
| Aber was ist, wenn Sie Code gemeinsam nutzen wollen?        | 355 |
| Header-Dateien gemeinsam nutzen                             | 356 |
| Objektdateien über den vollständigen Pfadnamen teilen       | 357 |
| Ein Archiv enthält .o-Dateien                               | 358 |
| Mit dem ar-Befehl ein Archiv erstellen ...                  | 359 |
| Schließlich die anderen Programme kompilieren               | 360 |
| Fit von Kopf bis Fuß expandiert                             | 365 |
| Kalorien berechnen                                          | 366 |
| Die Geschichte ist leider etwas komplizierter ...           | 369 |
| Programme bestehen aus vielen Teilen ...                    | 370 |
| Dynamisches Linken erfolgt zur Laufzeit                     | 372 |
| Kann man ein Archiv zur Laufzeit linken?                    | 373 |
| Erst die Objektdateien erstellen                            | 374 |
| Der Name Ihrer dynamischen Bibliothek ist plattformabhängig | 375 |
| Ihr C-Werkzeugkasten                                        | 387 |



# 2. C-Workshop

## OpenCV

Stellen Sie sich vor, Ihr Computer könnte ein Auge auf Ihr Haus werfen, während Sie unterwegs sind, und Sie informieren, wenn sich dort unbefugt jemand herumtreibt. Genau das ist mit einer an Ihren Rechner angeschlossenen oder in ihn eingebauten Webcam und den ausgefeilten Techniken von OpenCV möglich!



# Prozesse und Systemaufrufe

## Grenzverletzungen

9

### Es wird Zeit, über den Tellerrand hinauszublicken.

Sie haben bereits erfahren, dass Sie komplexe Anwendungen aufbauen können, indem Sie auf der Kommandozeile kleine Werkzeuge miteinander verbinden. Aber was ist, *wenn Sie andere Programme* aus Ihrem Code heraus nutzen wollen? In diesem Kapitel werden Sie lernen, wie Sie **Systemeinrichtungen** nutzen, um **Prozesse** zu erstellen und zu kontrollieren. Das wird Ihren Programmen Zugriff auf *E-Mail*, das *Web* und Unmengen anderer *Werkzeuge* geben, die Sie installiert haben. Wenn Sie das Ende dieses Kapitels erreicht haben, werden Sie die Macht haben, die **Grenzen von C** zu verlassen.

|                                                                                            |     |
|--------------------------------------------------------------------------------------------|-----|
| Systemaufrufe sind Ihr heißer Draht zum Betriebssystem                                     | 398 |
| Dann bricht jemand in das System ein ...                                                   | 402 |
| Sicherheit ist nicht das einzige Problem                                                   | 403 |
| Die <code>exec()</code> -Funktionen bieten Ihnen mehr Kontrolle                            | 404 |
| Es gibt diverse <code>exec()</code> -Funktionen                                            | 405 |
| Die Array-Funktionen: <code>execv()</code> , <code>execvp()</code> , <code>execve()</code> | 406 |
| Umgebungsvariablen übergeben                                                               | 407 |
| Die meisten Systemaufrufe scheitern auf gleiche Weise                                      | 408 |
| Nachrichten lesen mit RSS                                                                  | 416 |
| <code>exec()</code> ist die Ziellinie für Ihr Programm                                     | 420 |
| Mit <code>fork()</code> + <code>exec()</code> einen Kindprozess starten                    | 421 |
| Ihr C-Werkzeugkasten                                                                       | 427 |





10

Interprozesskommunikation

Ein nettes Gespräch

Prozesse machen ist nur die halbe Miete.

Was ist, wenn Sie den Prozess *steuern* wollen, nachdem er gestartet ist? Was, wenn Sie ihm *Daten senden* wollen? Oder seine *Ausgabe lesen*? Die **Interprozesskommunikation** ermöglicht Prozessen, gemeinsame Sache zu machen. Wir werden Ihnen zeigen, wie Sie die **Macht** Ihres Codes multiplizieren, indem Sie ihn mit anderen Programmen auf Ihrem Systemen **reden lassen**.

|                                                |     |
|------------------------------------------------|-----|
| Eingabe und Ausgabe umleiten                   | 430 |
| Ein Blick in einen typischen Prozess           | 431 |
| Eine Umleitung ersetzt einfach die Datenströme | 432 |
| fileno() nennt Ihnen den Dateideskriptor       | 433 |
| Manchmal muss man warten ...                   | 438 |
| Mit dem Kind verbunden bleiben                 | 442 |
| Prozesse mit Pipes verbinden                   | 443 |
| Fallstudie: Storys in einem Browser öffnen     | 444 |
| Im Kind                                        | 445 |
| Im Elternprozess                               | 445 |
| Eine Webseite in einem Browser öffnen          | 446 |
| Der Tod eines Prozesses                        | 451 |
| Signale abfangen und eigenen Code ausführen    | 452 |
| sigactions werden mit sigaction() registriert  | 453 |
| Den Code den Signal-Handler nutzen lassen      | 454 |
| Mit kill Signale senden                        | 457 |
| Ihrem Code einen Weckruf senden                | 458 |
| Ihr C-Werkzeugkasten                           | 466 |



```
#include <stdio.h>

int main()
{
    char name[30];
    printf("Ihr Name: ");
    fgets(name, 30, stdin);
    printf("Hallo %s\n", name);
    return 0;
}
```

Datei Bearbeiten Fenster Hilfe

> ./gruss

Ihr Name: ^C

>

Wenn Sie Strg-C drücken, stellt das Programm die Arbeit ein. Aber warum?

xx

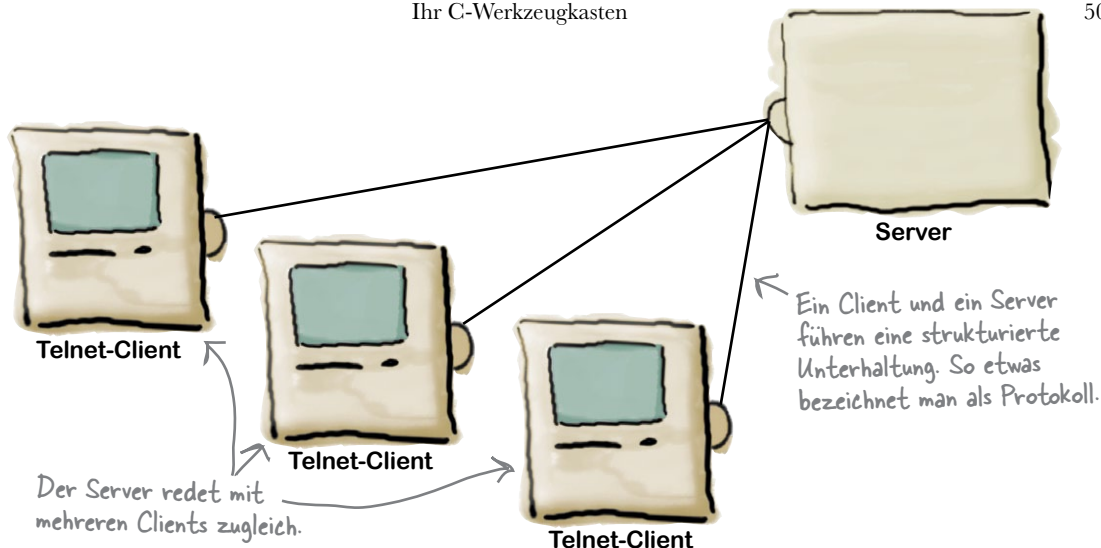
## 11

## Sockets und Netzwerke

**127.0.0.1 ist ein toller Ort****Programme auf unterschiedlichen Systemen müssen miteinander reden.**

Sie haben gelernt, wie Sie mithilfe von I/O-Operationen mit Dateien reden und wie Prozesse auf dem gleichen System miteinander kommunizieren können. Jetzt werden Sie nach *dem Rest der Welt greifen* und erfahren, wie man C-Programme schreibt, die **über das Netzwerk** und **quer über die ganze Welt** mit anderen Programmen reden können. Wenn Sie das Ende des Kapitels erreicht haben, werden Sie dazu in der Lage sein, **Programme zu schreiben, die sich wie Server verhalten, und Programme, die sich wie Clients verhalten.**

|                                                          |     |
|----------------------------------------------------------|-----|
| Der Internet-Knock-Knock-Server                          | 468 |
| Knock-Knock-Server im Überblick                          | 469 |
| BLAB: Wie Server mit dem Internet reden                  | 470 |
| Ein Socket ist kein billiger Datenstrom                  | 472 |
| Manchmal startet der Server nicht korrekt                | 476 |
| Warum Mama gepredigt hat, immer auf die Fehler zu achten | 477 |
| Vom Client lesen                                         | 478 |
| Der Server kann immer nur mit jeweils einer Person reden | 485 |
| Sie können einen eigenen Prozess für jeden Client forken | 486 |
| Einen Webclient schreiben                                | 490 |
| Clients an die Macht                                     | 491 |
| Ein Socket für eine IP-Adresse erstellen                 | 492 |
| getaddrinfo() ermittelt Adressen zu Domains              | 493 |
| Ihr C-Werkzeugkasten                                     | 500 |



## 12

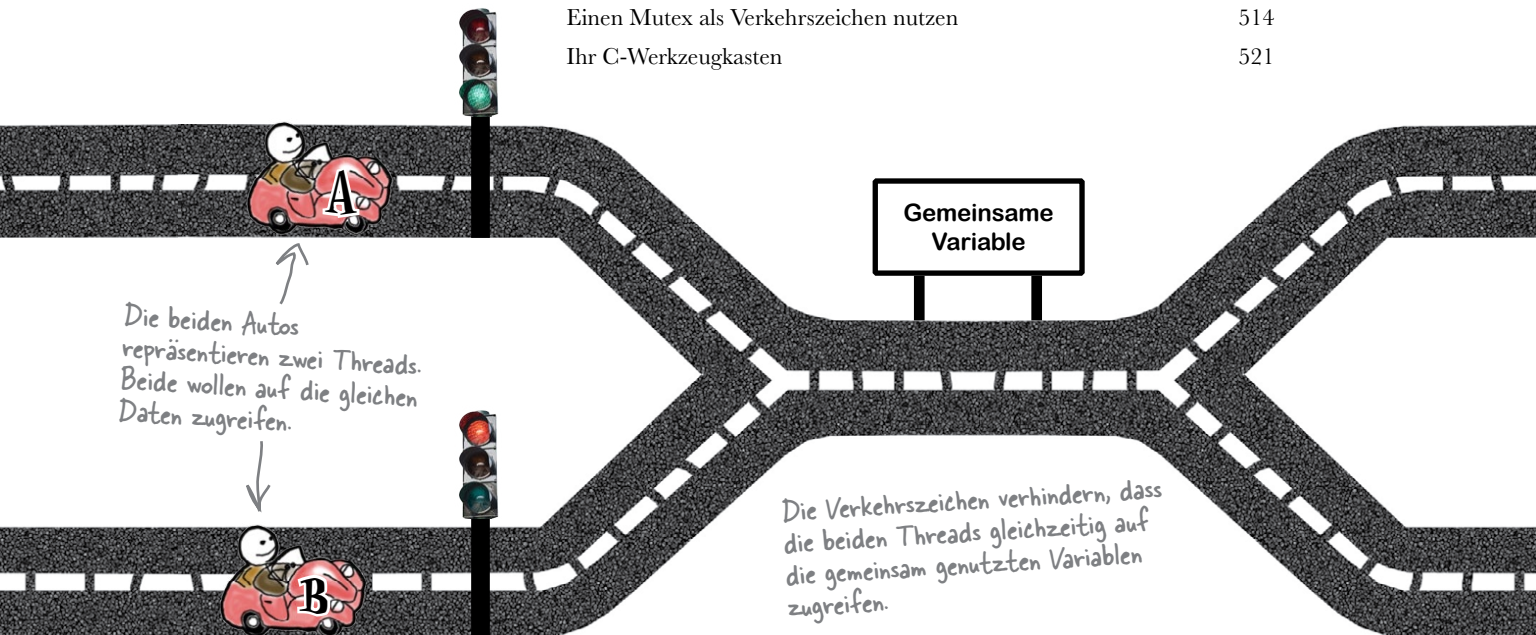
## Threads

## Parallelwelten

Häufig müssen Programme mehrere Dinge gleichzeitig tun.

Mit POSIX-Threads können Sie Ihren Code reaktionsfähiger machen, indem Sie **verschiedene Codeteile abspalten und parallel laufen lassen**. Doch aufgepasst! Threads sind mächtige Werkzeuge, und Sie sollten tunlichst vermeiden, dass sie sich in die Quere kommen. In diesem Kapitel werden Sie lernen, wie Sie die **Verkehrsschilder** und **Straßenmarkierungen** einrichten, die **Codeunfälle verhindern**. Am Ende werden Sie wissen, wie man **POSIX-Threads erstellt** und wie man **Synchronisationsmechanismen nutzt**, um **die Integrität wichtiger Daten zu sichern**.

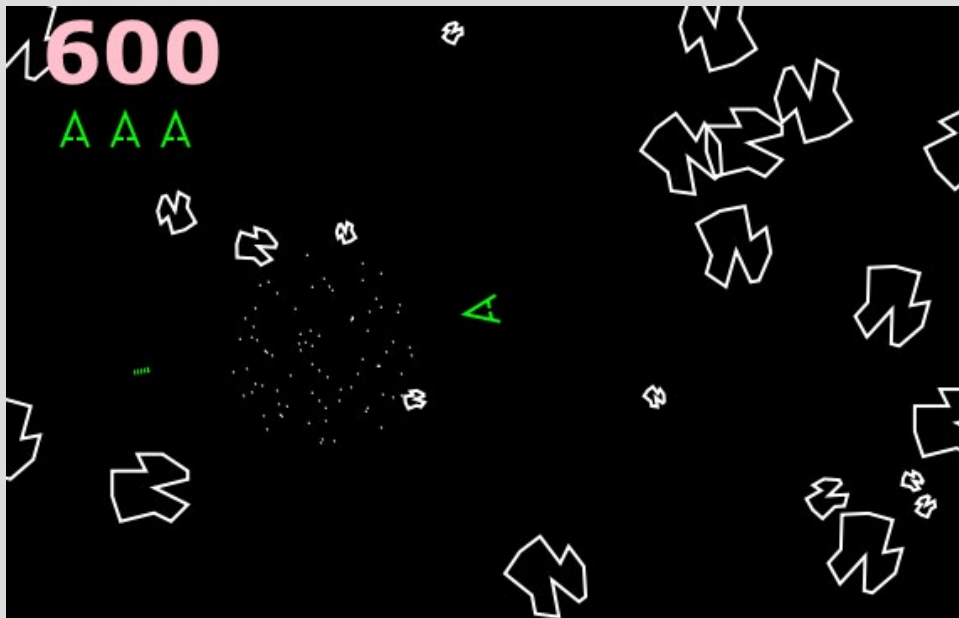
|                                                              |     |
|--------------------------------------------------------------|-----|
| Aufgaben verrichtet man nacheinander ... oder auch nicht ... | 502 |
| ... und Prozesse sind nicht immer die Antwort                | 503 |
| Einfache Prozesse tun eine Sache nach der anderen            | 504 |
| Zusätzliches Personal: Threads                               | 505 |
| Wie man Threads erstellt?                                    | 506 |
| Threads mit pthread_create erstellen                         | 507 |
| Der Code ist nicht Thread-sicher                             | 512 |
| Sie müssen Verkehrszeichen einführen                         | 513 |
| Einen Mutex als Verkehrszeichen nutzen                       | 514 |
| Ihr C-Werkzeugkasten                                         | 521 |



# 3. C-Workshop

## Blasteroids

Einer der wichtigsten Gründe dafür, dass die Leute in C programmieren lernen wollen, ist, dass sie dann Spiele schreiben können. In diesem Workshop werden wir einem der beliebtesten und langlebigsten Videospiele aller Zeiten Tribut zollen. Schreiben wir Blasteroids!



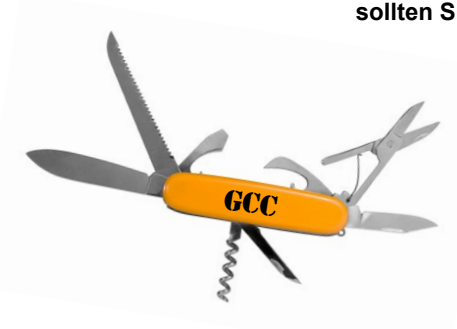
Anhang A: Was übrig bleibt

Die Top Ten der Dinge (die wir nicht behandelt haben)

A

Wir haben viel behandelt, und doch bleibt manches offen.

Da sind noch ein paar Dinge, die Sie wahrscheinlich wissen sollten. Wir würden uns nicht wohlfühlen, wenn wir sie nicht erwähnten, auch wenn wir sie nur kurz anreißern können – weil wir Ihnen ein Buch geben wollten, das Sie auch ohne ausgiebiges Muskeltraining noch transportieren können. Bevor Sie das Buch zu den Akten legen, sollten Sie sich also noch diese Kleinigkeiten ansehen.



|                             |     |
|-----------------------------|-----|
| 1. Operatoren               | 540 |
| 2. Präprozessordirektiven   | 542 |
| 3. Das Schlüsselwort static | 543 |
| 4. Wie groß die Dinge sind  | 544 |
| 5. Automatisierte Tests     | 545 |
| 6. Mehr zu gcc              | 546 |
| 7. Mehr zu make             | 548 |
| 8. Entwicklungswerkzeuge    | 550 |
| 9. GUIs erstellen           | 551 |
| 10. Referenzmaterial        | 552 |

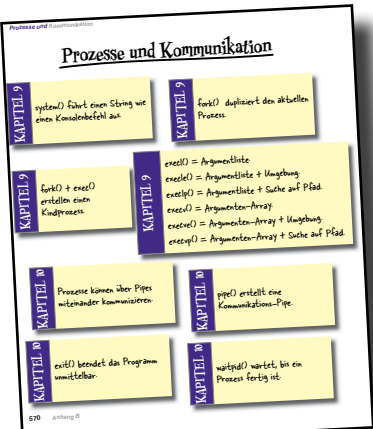
Anhang B: C-Themen

Zusammenfassungen im Überblick

B

Träumen Sie auch davon, das gesamte C-Wissen wäre an einem einzigen Ort vorzufinden?

Das ist eine Zusammenfassung aller C-Themen und Prinzipien, die wir in diesem Buch behandelt haben. Werfen Sie einen Blick darauf und schauen Sie, ob Sie sich alle eingeprägt haben. Neben den Merksätzen steht jeweils das Kapitel, aus dem sie kommen, damit Sie leicht an die entsprechende Stelle zurückkehren können, um Ihrem Gedächtnis auf die Sprünge zu helfen. Vielleicht sollten Sie diese Seiten sogar herausschneiden und bei sich an die Wand hängen.



## 4 Mehrere Quelldateien

# Zerlegen und zusammenbauen



**Wenn Sie ein großes Programm erstellen, heißt das nicht, dass Sie auch eine große Quelldatei haben wollen.**

Können Sie sich vorstellen, wie schwierig und zeitaufwendig die Wartung einer einzigen Quelldatei bei umfangreichen Programmen werden kann? In diesem Kapitel werden Sie erfahren, wie Ihnen C ermöglicht, Quellcode in **kleine, handhabbare Happen** zu zerlegen und diese dann zu **einem großen Programm** zusammenzusetzen. Auf dem Weg dorthin werden Sie etwas mehr über die **Feinheiten von Datentypen** erfahren und werden jemandem über den Weg laufen, der einer Ihrer besten Freunde werden wird: **make**.



Die Anzahl an Bauteilen  
in einer Rakete

Die Menge Treibstoff, die  
die Rakete benötigt (Liter)



## Erraten Sie den Datentyp

C kann mit einer ganzen Reihe unterschiedlicher Datentypen umgehen: Zeichen, ganzen Zahlen, Gleitkommazahlen für Alltagswerte, Gleitkommazahlen für präzise wissenschaftliche Berechnungen. Ein paar dieser Datentypen werden auf der gegenüberliegenden Seite aufgeführt. Schauen Sie, ob Sie herausfinden können, welcher Datentyp für die jeweiligen Beispiele benötigt wird.

**Beachten Sie:** Alle Beispiele nutzen unterschiedliche Datentypen.



Die Entfernung der  
Startrampe zum Stern  
Proxima Centauri  
(Lichtjahre)

Die Anzahl an Sternen  
im Universum, die wir  
nicht besuchen werden

Die Anzahl von  
Minuten bis zum Start

Die Buchstaben auf der  
Countdown-Anzeige



Das sind Zahlen,  
die Nachkomma-  
stellen enthalten.

### Gleitkomma

float

double

### Ganzzahl

short

long

int

char

Das ist richtig!  
In C werden Zeichen  
über ihre Zeichencodes  
gespeichert. Das heißt,  
dass auch sie eigentlich  
nur Zahlen sind!

Die Anzahl an Bauteilen in  
einer Rakete

int

Die Menge Treibstoff, die  
die Rakete benötigt (Liter)

float



### Erraten Sie den Datentyp, Lösung

C kann mit einer ganzen Reihe unterschiedlicher Datentypen umgehen: Zeichen, ganzen Zahlen, Gleitkommazahlen für Alltagswerte, Gleitkommazahlen für präzise wissenschaftliche Berechnungen. Ein paar dieser Datentypen werden auf der gegenüberliegenden Seite aufgeführt. Sie sollten herausfinden, welcher Datentyp für die jeweiligen Beispiele benötigt wird.

**Denken Sie daran:** Alle Beispiele nutzen unterschiedliche Datentypen.

Die Entfernung der  
Startrampe zum Stern  
Proxima Centauri (Lichtjahre)

double

Die Anzahl an Sternen  
im Universum, die wir  
nicht besuchen werden

long

Die Anzahl von  
Minuten bis zum Start

short

Die Buchstaben auf der  
Countdown-Anzeige

char

90:00  
minuten

Sehen wir uns an, warum ...

## Datentypen-Schnellkurs

### char

Jedes Zeichen wird im Speicher des Computers als Zeichencode gespeichert. Und der ist nur eine Zahl. Wenn der Computer ein A sieht, ist das für ihn das Gleiche, als sähe er die Zahl 65.

### int

↖ 65 ist der ASCII-Code für A.

Wenn Sie eine ganze Zahl speichern müssen, können Sie in der Regel einfach `int` nutzen. Wie groß ein `int` werden kann, ist nicht immer gleich, aber er umfasst mindestens 16 Bit. Im Allgemeinen kann `int` Zahlen bis zu ein paar Millionen speichern.

### short

Aber manchmal möchten Sie etwas Speicher sparen. Warum sollten Sie einen `int` nutzen, wenn Sie nur Zahlen speichern wollen, die lediglich ein paar Hundert oder Tausend betragen? Dafür ist `short` gedacht. Ein `short` braucht normalerweise halb so viel Platz wie ein `int`.

### long

Aber was ist, wenn Sie eine **echt große Anzahl** speichern wollen? Dafür hat man sich den Datentyp `long` ausgedacht. Auf manchen Systemen kann der Datentyp `long` *doppelt* so viel Speicher einnehmen wie ein `int` und kann Zahlen bis in die **Milliarden** festhalten. Aber weil die meisten Computer mit großen `ints` umgehen können, ist `long` auf vielen Systemen *genauso groß wie* `int`. Für einen `long` stehen mindestens 32 Bit bereit.

### float

`float` ist der grundlegende Datentyp für die Speicherung von Gleitkommawerten. Bei den meisten im Alltag benötigten Gleitkommazahlen – wie der Menge Flüssigkeit in einem Kölschglas – können Sie `float` verwenden.

### double

Und was ist, wenn es echt **genau** sein muss? Wenn Sie Berechnungen durchführen wollen, die auf eine große Anzahl von **Nachkommastellen** genau sein müssen, sollten Sie einen `double` nutzen. Ein `double` nimmt im Speicher doppelt so viel Platz ein wie ein `float` und nutzt den zusätzlichen Raum, um Zahlen zu speichern, die *größer und genauer* sind.

# Großes darf man nicht in Kleines stecken

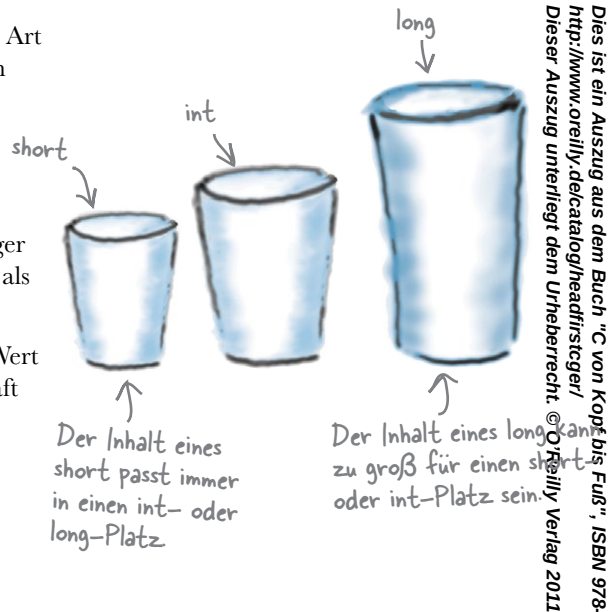
Wenn Sie Werte herumreichen, müssen Sie darauf achten, dass die Art des Werts der Art von Variablen entspricht, in der Sie ihn speichern werden.

Unterschiedliche Datentypen benötigen unterschiedlich viel Speicherplatz. Sie müssen also aufpassen, dass Sie nicht versuchen, einen Wert zu speichern, der zu groß für den Speicherplatz ist, der für eine Variable reserviert wird. `short`-Variablen brauchen weniger Speicherplatz als `ints`, und `ints` brauchen weniger Speicherplatz als `longs`.

Es entsteht also kein Problem, wenn Sie versuchen, einen `short`-Wert in einer `int`- oder `long`-Variablen zu speichern. Es gibt massenhaft Platz im Speicher, und der Code wird ordentlich funktionieren:

```
short x = 15;
int y = x;
printf("Der Wert von y = %i\n", y);
```

Das sagt, dass y = 15 ist.



Probleme treten auf, wenn Sie den umgekehrten Weg einschlagen – wenn Sie beispielsweise versuchen, einen `int`-Wert in einem `short` zu speichern:

```
int x = 100000;
short y = x;
print("Der Wert von y = %hi\n", y);
```

%hi ist der richtige Formatcode für einen short-Wert.

Manchmal kann der Compiler erkennen, dass Sie versuchen, einen zu großen Wert in einer zu kleinen Variablen zu speichern, und meldet dann eine Warnung. Aber häufig ist der Compiler nicht so schlau und kompiliert den Code ohne Beschwerden. Wenn Sie dann versuchen, den Code auszuführen, kann der Computer die Zahl 100.000 nicht in einer `short`-Variablen speichern. Der Computer wird so viele Einsen und Nullen reinpacken wie möglich, aber die Zahl, die in der Variablen `y` gespeichert wird, wird sich *erheblich* von der unterscheiden, die Sie speichern wollten:

```
Der Wert von y = -31072
```



## Freak-Futter

Warum erhalten wir eine negative Zahl, wenn wir versuchen, eine zu große Zahl in einem `short` zu speichern? Zahlen werden binär gespeichert. So sieht 100.000 binär aus:

```
x <- 0001 1000 0110 1010 0000
```

Wenn aber der Computer versucht, diesen Wert in einem `short` zu speichern, stehen ihm nur zwei Byte für die Speicherung zur Verfügung. Das Programm speichert nur die rechte Seite der Zahl:

```
y <- 1000 0110 1010 0000
```

**Vorzeichenbehaftete** Werte, deren Binärdarstellung im höchstwertigen Bit eine 1 haben, werden als negative Zahlen betrachtet. Und dieser verkürzte Wert entspricht dieser Dezimalzahl:

```
-31072
```

# Mit Casts floats in ganze Zahlen packen

Was, denken Sie, wird der folgende Code anzeigen?

```
int x = 7;
int y = 2;
float z = x / y;
printf("z = %f\n", z);
```

Die Antwort? **3,0000**. Warum das? `x` und `y` sind beide `ints`, also ganze Zahlen, und wenn Sie zwei davon teilen, erhalten Sie immer eine aufgerundete ganze Zahl – hier **3**.

Was aber ist, wenn Sie bei einer Berechnung mit ganzen Zahlen ein Gleitkommaergebnis erhalten wollen? Sie könnten die ganzen Zahlen zuvor in `float`-Variablen speichern, aber das wäre etwas umständlich. Stattdessen können Sie einen **Cast** nutzen, um die Zahlen im Vorübergehen umzuwandeln:

```
int x = 7;
int y = 2;
float z = (float)x / (float)y;
printf("z = %f\n", z);
```

Das **(float)** *castet* einen `int`-Wert auf einen `float`-Wert. Die Berechnung funktioniert dann genau so, als würden die ganze Zeit Gleitkommazahlen verwendet. Wenn der Compiler sieht, dass Sie eine Addition, Subtraktion, Multiplikation oder Division mit einem Gleitkommawert und einem ganzzahligen Wert vornehmen, *castet* er automatisch die Zahlen für Sie. Das heißt, dass Sie die Anzahl expliziter Casts in Ihrem Code reduzieren können:

```
float z = (float)x / y; ← Der Compiler castet y automatisch auf einen float.
```

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.o-reilly.com  
Dieser Auszug  
O'Reilly Verlag 2011



Sie können Datentypen einige weitere Schlüsselwörter hinzufügen, die beeinflussen, wie Zahlen interpretiert werden:

## unsigned

Die Zahl ist immer positiv. Weil sie sich nicht darum kümmern müssen, negative Zahlen festzuhalten, können `unsigned`-Zahlen größere Zahlen aufnehmen, da ein Bit mehr zur Speicherung zur Verfügung steht. Ein `unsigned int` speichert also Zahlen von 0 bis zu einem Maximum, das doppelt so groß ist wie die größte Zahl, die in einem `int` gespeichert werden kann. Es gibt auch das Schlüsselwort `signed`, das aber fast nie verwendet wird, da es bei allen Datentypen der Standard ist.

```
unsigned char c;
```

Das speichert wahrscheinlich Zahlen von 0 bis 255.

## long

Sie sehen richtig: Sie können einem Datentyp das Wort `long` voranstellen, um ihn länger zu machen. Ein `long int` ist also eine längere Version eines `int`. Das heißt, dass er einen größeren Bereich von Zahlen speichern kann. Und ein `long long` ist länger als ein `long`. Sie können `long` auch für Gleitkommazahlen verwenden.

→ `long double d;`  
Eine so RICHTIG genaue Zahl.

long long gibt es nur in C99 und C11.



Hier ist ein neues Programm, das den Kellnern im Von Kopf bis Fuß-Restaurant hilft, die Tischrechnung zu erstellen. Der Code rechnet automatisch die Beträge zusammen und schlägt die Mehrwertsteuer auf. Versuchen Sie, die Lücken zu füllen.

**Hinweis:** In diesem Programm könnten unterschiedliche Datentypen genutzt werden, aber welche würden Sie für die Art von Beträgen einsetzen, die Sie hier erwarten?

```
#include <stdio.h>

.....summe = 0.0;
.....anzahl = 0;
.....steuersatz = 19;

.....mit_steuer_addieren(float f)
{
    .....steuer = 1 + steuersatz / 100 ..... ;
    summe = summe + (f * steuer);
    anzahl = anzahl + 1;
    return summe;
}

int main()
{
    .....wert;
    printf("Betrag für Posten: ");
    while (scanf("%f", &wert) == 1) {
        printf("Zwischensumme: %.2f\n", mit_steuer_addieren(wert));
        printf("Betrag für Posten: ");
    }
    printf("\nGesamtsumme: %.2f\n", summe);
    printf("Anzahl Posten: %hi\n", anzahl);
    return 0;
}
```

*Handwritten notes:*

- Arrow pointing to `%.2f` in the first `printf` call: `%.2f` formatiert eine Gleitkommazahl auf zwei Nachkommastellen.
- Arrow pointing to `%hi` in the `printf` call: `%hi` wird zur Formatierung von shorts genutzt.





## LÖSUNG ZUR ÜBUNG

Hier ist ein neues Programm, das den Kellnern im Von Kopf bis Fuß-Restaurant hilft, die Tischrechnung zu erstellen. Der Code rechnet automatisch die Beträge zusammen und schlägt die Mehrwertsteuer auf. Sie sollten versuchen, die Lücken zu füllen.

**Hinweis:** In diesem Programm könnten unterschiedliche Datentypen genutzt werden, aber welche würden Sie für die Art von Beträgen einsetzen, die Sie hier erwarten?

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

Hier brauchen Sie eine kleine Gleitkommazahl für den Betrag.

```
.....float..... summe = 0.0;
.....short..... anzahl = 0;
.....short..... steuersatz = 19;
```

Eine Rechnung hat nur wenige Posten, wir beschränken uns deswegen auf einen short.

```
.....float..... mit_steuer_addieren(float f)
```

Wir liefern eine kleine Menge Bargeld zurück; das ist also ein float.

Für diesen Bruch ist ein float in Ordnung.

```
{
.....float..... steuer = 1 + steuersatz / 100 .....0.....;
    summe = summe + (f * steuer);
    anzahl = anzahl + 1;
    return summe;
}
```

Wenn Sie .0 anhängen, sorgen Sie dafür, dass die Berechnung float-basiert ist. Hätten Sie 100 stehen gelassen, hätten Sie eine ganze Zahl zurückerhalten.

```
int main()
```

```
{
.....float..... wert;
    printf("Betrag für Posten: ");
    while (scanf("%f", &wert) == 1) {
        printf("Zwischensumme: %.2f\n", mit_steuer_addieren(wert));
        printf("Betrag für Posten: ");
    }
    printf("\nGesamtsumme: %.2f\n", summe);
    printf("Anzahl Posten: %hi\n", anzahl);
    return 0;
}
```

Alle Preise passen locker in einen float.

$1 + \text{steuersatz} / 100$ ; würde den Wert 1 liefern, weil bei einer ganzzahligen Berechnung  $6/100 == 0$  gilt.

# Datentypgrößen unter der Lupe



Datentypen haben auf unterschiedlichen Plattformen unterschiedliche Größen. Wie aber bringen Sie in Erfahrung, wie groß ein `int` ist oder wie viele Bytes ein `double` einnimmt? Glücklicherweise bietet die C-Standardbibliothek einige Header, die die entsprechenden Informationen bieten. Folgendes Programm informiert Sie über die Größe von `ints` und `floats`:

```
#include <stdio.h>
#include <limits.h> ← Enthält die Werte für ganzzahlige Typen wie int und char.
#include <float.h> ← Enthält die Werte für floats und doubles.
```

```
int main()
{
    printf("Der Wert von INT_MAX ist %i\n", INT_MAX);
    printf("Der Wert von INT_MIN ist %i\n", INT_MIN);
    printf("Ein int hat %z Bytes\n", sizeof(int));
    printf("Der Wert von FLT_MAX ist %f\n", FLT_MAX);
    printf("Der Wert von FLT_MIN ist %.50f\n", FLT_MIN);
    printf("Ein float hat %u Bytes\n", (unsigned int) sizeof(float));
    return 0;
}
```

Wenn Sie diesen Code kompilieren und ausführen, werden Sie eine Ausgabe folgender Form erhalten:

[illegible]

Die Werte, die Sie auf Ihrem Rechner sehen, werden wahrscheinlich anders lauten.

Was ist, wenn Sie Informationen zu chars oder doubles suchen? Oder longs? Kein Problem. Ersetzen Sie einfach INT und FLT durch CHAR (char), DBL (double), SHRT (short) oder LNG (long).

ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
www.oreilly.de/catalog/headfirstcger/  
Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

## Es gibt keine Dummen Fragen

**F:** Warum sind die Datentypen nicht bei allen Betriebssystemen gleich? Wäre es nicht weniger verwirrend, wenn sie überall gleich wären?

**A:** C nutzt unterschiedliche Datentypen auf den verschiedenen Betriebssystemen und Prozessoren, damit die Hardware sie optimal ausnutzen kann.

**F:** In welcher Weise?

**A:** Als C geschaffen wurde, basierten die meisten Systeme auf 8 Bit, heute sind es 32 oder 64 Bit. Weil C die Größe der Datentypen nicht genau vorgibt, konnte es mit der Zeit gehen. Und wenn neue Systeme aufkommen, wird C auch das Beste aus ihnen herausholen können.

**F:** Was heißt »auf 8 Bit oder 64 Bit basieren« eigentlich?

**A:** Technisch gesehen, kann diese Zahl verschiedene Dinge bezeichnen, z. B. die Breite der CPU-Instruktionen oder die Datenmenge, die die CPU aus dem Speicher lesen kann. Diese Bitgröße gibt die bevorzugte Größe für die Zahlen an, mit denen der Computer arbeitet.

**F:** Und was hat das mit der Größe von `ints` und `doubles` zu tun?

**A:** Wenn ein Computer für die Arbeit mit 32-Bit-Zahlen optimiert ist, ist es sinnvoll, den grundlegenden Datentyp – `int` – 32 Bit breit zu machen.

**F:** Wie ganze Zahlen wie `ints` funktionieren, verstehe ich, aber wie werden `floats` und `doubles` gespeichert? Wie stellt der Computer eine Zahl mit Nachkommastellen dar?

**A:** Das ist kompliziert. Die meisten Computer nutzen einen Standard, der von der IEEE veröffentlicht wurde (<http://tinyurl.com/6defkv6>).

**F:** Muss ich verstehen, wie Gleitkommazahlen funktionieren?

**A:** Nein. Die große Mehrheit der Entwickler nutzt `floats` und `doubles`, ohne sich über die Details Gedanken zu machen.

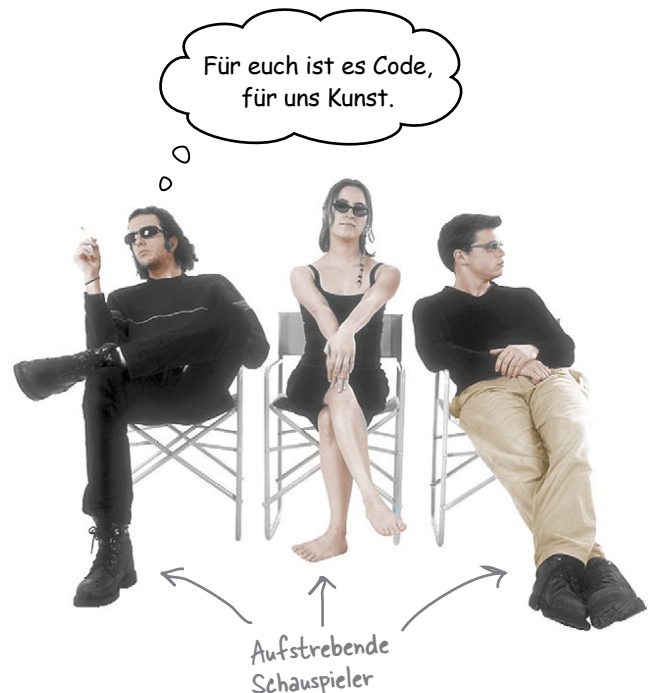
Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/headers/cger/>  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

## Oh nein, arbeitslose Schauspieler am Werk ...

Manche Menschen sind einfach nicht zum Programmierer geboren. Es scheint, dass einige aufstrebende Schauspieler rollenlose Zeiten *überbrücken*, indem sie ihre Börse damit aufbessern, nebenbei etwas Code zu kreieren. Sie haben beschlossen, etwas Zeit in die Aufbesserung des Codes für das Rechnungsprogramms zu stecken.

Einmal in die Mangel genommen, gefiel den Schauspielern der Code anschließend schon um einiges besser ... es gab nur ein winziges Problem.

**Der Code ließ sich nicht mehr kompilieren.**



## Schauen wir uns an, was dem Code widerfahren ist

Das haben die Schauspieler mit dem Code angestellt. Sie können sehen, dass sie eine Reihe von Dingen geändert haben.

```
#include <stdio.h>

float summe = 0.0;
short anzahl = 0;
/* 19%. Da nimmt mein Agent ja weniger ...*/
short steuersatz = 19;

int main()
{
    /* Künstlerisch betrachtet, hat das kaum Wert. */
    float wert;
    printf("Betrag für Posten: ");
    while (scanf("%f", &wert) == 1) {
        printf("Zwischensumme: %.2f\n", mit_steuer_addieren(wert));
        printf("Betrag für Posten: ");
    }
    printf("\nGesamtsumme: %.2f\n", summe);
    printf("Anzahl Posten: %hi\n", anzahl);
    return 0;
}

float mit_steuer_addieren(float f)
{
    float steuer = 1 + steuersatz / 100.0;
    /* Was ist mit Trinkgeld? Sprecherziehung kostet.*/
    summe = summe + (f * steuer);
    anzahl = anzahl + 1;
    return summe;
}
```

Dem Code wurden Kommentare hinzugefügt, und außerdem wurde die **Abfolge der Funktionen geändert**. Andere Modifizierungen wurden nicht vorgenommen.

Eigentlich sollte es also keine Probleme geben. Der Code müsste einsatzbereit sein. Alles war in Ordnung, bis sie daran gingen, **den Code zu kompilieren ...**

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.oreilly.de/catalog/headsrticger/  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011



# TESTLAUF

Wenn Sie die Konsole öffnen und versuchen, das Programm zu kompilieren, geschieht Folgendes:

```

Datei Bearbeiten Fenster Hilfe SchusterBleibBeiDeinenLeisten
> gcc rechnung.c -o rechnung && ./rechnung
rechnung.c: In function "main":
rechnung.c:14: warning: format "%.2f" expects type
"double", but argument 2 has type "int"
rechnung.c: At top level:
rechnung.c:23: error: conflicting types for "mit_steuer_addieren"
rechnung.c:14: error: previous implicit declaration of
"mit_steuer_addieren" was here
  
```

## Mist.

Das ist gar nicht gut. Was heißt dieses `error: conflicting types for 'mit_steuer_addieren'`? Was dieses *previous implicit declaration*? Und warum meint der Compiler, die Zeile, die die Gesamtsumme ausgibt, erhalte jetzt einen `int`? Sollte das nicht eigentlich ein `float` sein?

Der Compiler ignoriert die Änderungen, die an den Kommentaren vorgenommen wurden. Die sollten also eigentlich keinen Unterschied machen. Das heißt, dass das Problem **durch die Änderung der Abfolge der Funktionen** entstanden sein muss. Aber wenn die Abfolge das Problem ist, warum liefert der Compiler dann nicht einfach eine Meldung mit folgendem Gehalt:



Das ist ganz ernst gemeint. Warum unterstützt uns der Compiler hier nicht?

Wenn Sie genau verstehen wollen, was hier geschieht, müssen Sie sich eine Weile in den Compiler hineinversetzen und die Angelegenheit aus seiner Perspektive betrachten. Dann werden Sie feststellen, dass der Compiler vielleicht sogar versucht, *etwas zu hilfreich* zu sein.

# Compiler mögen keine Überraschungen

Was genau geschieht also, wenn der Compiler diese Codezeile sieht?

```
printf("Zwischensumme: %.2f\n", mit_steuer_addieren(wert));
```

## 1 Der Compiler sieht eine Funktion, die er nicht kennt.

Statt sich zu beschweren, denkt sich der Compiler, dass er später in der Datei wahrscheinlich mehr über die Funktion erfahren wird. Der Compiler merkt sich einfach, dass er weiter hinten in der Datei auf die Funktion achten muss. Genau das aber ist das Problem ...

Komisch. Ein Aufruf einer Funktion, von der ich noch nie gehört habe. Notieren wir uns die mal. Später werden wir sicher mehr erfahren.



## 2 Der Compiler muss wissen, welchen Datentyp die Funktion liefert.

Natürlich kann der Compiler noch nicht wissen, was die Funktion liefern wird, und stellt deswegen eine **Vermutung** an. Der Compiler geht davon aus, dass sie einen `int` liefert.

Ich wette, die Funktion liefert einen `int`. Tun ja die meisten.



## 3 Wenn er den Code für die eigentliche Funktion erreicht, liefert er die Meldung »conflicting types for 'mit\_steuer\_addieren'«.

Das liegt daran, dass der Compiler denkt, er habe zwei Funktionen gleichen Namens. Eine Funktion ist die echte Funktion in der Datei. Die andere ist die, die der Compiler erwartet, die, die einen `int` liefern sollte.

Eine Funktion namens `mit_steuer_addieren()`, die einen `float` liefert??? In meinen Aufzeichnungen steht, dass es bereits eine solche gibt, die einen `int` liefert...



**KOPF-  
NUSS**

Der Computer vermutet, dass die Funktion einen `int` liefert, obwohl sie eigentlich einen `float` liefert. Wie hätten Sie dieses Problem gelöst, wenn Sie der Erfinder von C wären?

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.oreilly.de/catalog/headers/cger/  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Ist doch schnuppe, Mensch! Mir ist absolut **egal**, wie C dieses Problem löst. Bring einfach die verdammten Funktionen in die richtige Reihenfolge!

**Sie könnten die Funktionen einfach in die richtige Reihenfolge bringen und die Funktion definieren, bevor Sie sie in main() aufrufen.**

Durch eine Änderung der Reihenfolge können Sie verhindern, dass der Compiler gefährliche Annahmen in Bezug auf den Rückgabewert unbekannter Funktionen macht. Aber wenn Sie immer dazu gezwungen wären, Funktionen in einer bestimmten Reihenfolge zu definieren, hätte das eine Reihe von Konsequenzen.

## Das Reparieren der Funktionsabfolge ist eine Qual

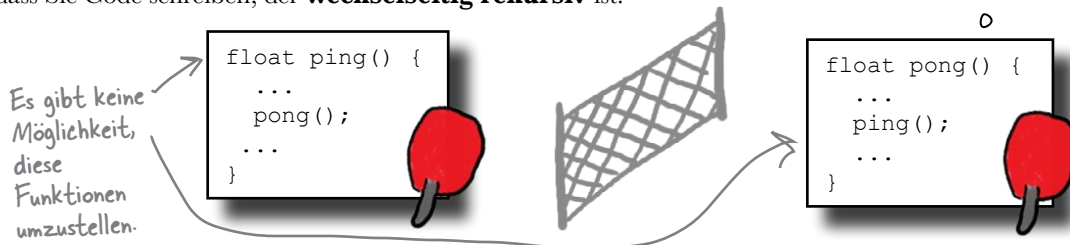
Angenommen, Sie fügen Ihrem Code eine coole neue Funktion hinzu, die alle umwerfend finden:

```
int irgendwas_tun(){...}
float was_fantastisches_tun(int beeindruckend) {...}
int das_richtige_tun() {
    was_fantastisches_tun(11);
}
```

Was passiert, wenn Sie *später* entscheiden, dass Ihr Programm sogar noch *besser* wird, wenn Sie in den `irgendwas_tun()`-Code einen `was_fantastisches_tun()`-Aufruf einbauen? Dann müssen Sie die **Funktion** in der Datei **nach oben verschieben**. Die meisten Programmierer wollen ihre Zeit mit der Optimierung dessen verbringen, was ihr Code leisten kann. Es wäre besser, wenn sie nicht die Abfolge ihres Codes anpassen müssten, um den Compiler glücklich zu machen.

## Manchmal gibt es keine richtige Reihenfolge

Die Situation ist zwar selten, aber gelegentlich kann es schon vorkommen, dass Sie Code schreiben, der **wechselseitig rekursiv** ist:



Wenn Sie zwei Funktionen haben, *die sich gegenseitig aufrufen*, wird eine **davon immer aufgerufen, bevor sie in der Datei definiert wird**.

Aus beiden Gründen wäre es wirklich nützlich, wenn man Funktionen in eben der Reihenfolge definieren könnte, die gerade am einfachsten ist. Aber wie?





## Die Deklaration von der Definition trennen

Erinnern Sie sich, dass sich der Compiler eine Notiz zu der Funktion machte, die er später in der Datei erwartete? Dass der Compiler derartige Annahmen überhaupt macht, können Sie vermeiden, indem Sie **ihm explizit sagen, welche Funktionen er erwarten soll**. Wenn Sie den Compiler über eine Funktion informieren, bezeichnet man das als **Funktionsdeklaration**:

Die Deklaration sagt dem Compiler, welchen Rückgabewert er erwarten soll.

→ `float mit_steuer_addieren();`

← Eine Deklaration enthält keinen Funktionsinhalt.  
← Sie endet einfach mit einem ; (Semikolon).

Die Deklaration ist einfach nur eine **Funktionssignatur**: eine Notiz, die sagt, wie eine Funktion heißt, welche Art Parameter sie erwartet und **welchen Typ an Daten sie zurückliefert**.

Haben Sie eine Funktion deklariert, muss der Compiler keine Annahmen mehr machen, und es gibt auch keine Probleme mehr, wenn Sie eine Funktion erst nach der ersten Verwendung definieren.

Wenn Ihr Code viele Funktionen enthält und wenn Sie sich keine Gedanken über deren Abfolge in der Datei machen wollen, können Sie Ihrem C-Programmcode eine Liste von Funktionsdeklarationen voranstellen:

```
float was_fantastisches_tun();
double beeindruckend_2_punkt_0();
int schwarzer_peter();
char margarita_mixen(int zahl);
```

Deklarationen haben keinen Körper.

Noch besser ist, dass C es Ihnen ermöglicht, ganze Sätze von Deklarationen *aus Ihrem Code herauszuziehen* und in eine **Header-Datei** zu stecken. Header-Dateien haben Sie bereits genutzt, um Code aus der C-Standardbibliothek einzufügen:

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

← Diese Zeile schließt den Inhalt der Header-Datei mit dem Namen `stdio.h` ein.

**Schauen wir uns an, wie man eigene Header-Dateien erstellt.**



Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/headsinger/>  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

# Ihre erste Header-Datei

Zur Erstellung eines Headers müssen Sie nur **zwei Dinge** tun:

- 1 **Erstellen Sie eine neue Datei mit der Dateinamenserweiterung .h.**  
Wenn Sie ein Programm namens *rechnung* aufbauen, erstellen Sie eine Datei namens *rechnung.h* und stecken in diese Ihre Deklarationen:

```
float mit_steuer_addieren(float f);
```



rechnung.h

Die Funktion `main()` müssen Sie nicht in die Header-Datei einbauen, da sie nie von anderem Code aufgerufen wird.

- 2 **Fügen Sie die Header-Datei in das Programm ein.**  
Zu Anfang Ihres Programms sollten Sie eine weitere `include`-Zeile ergänzen:

Fügen Sie diesen Include Ihren anderen Include-Zeilen hinzu.

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



rechnung.c

Wenn Sie den Namen der Header-Datei schreiben, müssen Sie darauf achten, dass Sie ihn in doppelte Anführungszeichen einschließen und nicht in spitze Klammern. Warum dieser Unterschied? Wenn der Compiler einen `include` mit spitzen Klammern sieht, geht er davon aus, dass er die Header-Datei in einem der Verzeichnisse findet, in denen Bibliothekscode gespeichert ist. Aber **Ihre** Header-Datei befindet sich im gleichen Verzeichnis wie Ihre **.c**-Datei. Wenn Sie den Header-Namen in Anführungszeichen setzen, sagen Sie dem Compiler, dass er nach einer lokalen Datei suchen soll.

← Lokale Header-Dateien können auch Verzeichnisnamen enthalten, aber üblicherweise stecken Sie sie in das gleiche Verzeichnis wie die C-Datei.

Wenn der Compiler das `#include` im Code antrifft, liest er den Inhalt der Header-Datei ein, als wäre er in den Code eingegeben worden.

Lagern Sie Ihre Deklarationen in eine separate Header-Datei aus, hält das nicht nur Ihren Code kürzer, sondern hat außerdem *einen weiteren Vorteil*, den Sie in ein paar Seiten erfahren werden.

Zunächst aber wollen wir uns ansehen, ob eine Header-Datei das Problem löst.

## #include ist eine Präprozessor-Instruktion.



# TESTLAUF

Wenn Sie den Code jetzt kompilieren, geschieht Folgendes:

Keine Fehler-  
meldungen  
mehr.

```
Datei Bearbeiten Fenster Hilfe MitKöpfchen
> gcc rechnung.c -o rechnung
```

Der Compiler liest die Funktionsdeklaration aus der Header-Datei und muss nicht mehr raten, welchen Rückgabebetyp die Funktion hat. Die Abfolge der Funktionen spielt keine Rolle mehr.

Um zu prüfen, ob alles in Ordnung ist, können Sie das generierte Programm testen, um sich zu versichern, dass es genau so funktioniert wie zuvor.

```
Datei Bearbeiten Fenster Hilfe MitKöpfchen
> ./rechnung
Betrag für Posten: 1.23
Zwischensumme: 1.30
Betrag für Posten: 4.57
Zwischensumme: 6.15
Betrag für Posten: 11.92
Zwischensumme: 18.78
Betrag für Posten: ^D
Gesamtsumme: 18.78
Anzahl Posten: 3
```

Drücken Sie hier Strg-D, damit das Programm nicht weiter nach Beträgen fragt.



## Spielen Sie Compiler

Schauen Sie sich das Programm unten an.  
Ein Teil fehlt. Spielen Sie also Compiler  
und sagen Sie, was Sie tun würden,  
wenn die jeweiligen auf der  
rechten Seite aufgelisteten  
Kandidatenfragmente in die  
Lücke eingesetzt würden.

Hier kommt der  
Kandidatencode rein.

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

```
printf("Auf dem Merkur hat ein Tag %f Stunden\n", tag);  
return 0;
```

```
}
```

```
float merkurtag_in_erdtagen()
```

```
{
```

```
    return 58.65;
```

```
}
```

```
int stunden_in_erdtag()
```

```
{
```

```
    return 24;
```

```
}
```

Hier sind die Codefragmente. ↗

↙ Setzen Sie Ihre Haken in die richtigen Kästen.

```
float merkurtag_in_erdtagen();
int stunden_in_erdtage();

int main()
{
    float taglaenge = merkurtag_in_erdtagen();
    int stunden = stunden_in_erdtage();
    float tag = taglaenge * stunden;
```

- ☐ Code kompilierbar.
- ☐ Warnung anzeigen.
- ☐ Programm funktioniert.

```
float merkurtag_in_erdtagen();

int main()
{
    float taglaenge = merkurtag_in_erdtagen();
    int stunden = stunden_in_erdtage();
    float tag = taglaenge * stunden;
```

- ☐ Code kompilierbar.
- ☐ Warnung anzeigen.
- ☐ Programm funktioniert.

```
int main()
{
    float taglaenge = merkurtag_in_erdtagen();
    int stunden = stunden_in_erdtage();
    float tag = taglaenge * stunden;
```

- ☐ Code kompilierbar.
- ☐ Warnung anzeigen.
- ☐ Programm funktioniert.

```
float merkurtag_in_erdtagen();
int stunden_in_erdtage();

int main()
{
    int taglaenge = merkurtag_in_erdtagen();
    int stunden = stunden_in_erdtage();
    float tag = taglaenge * stunden;
```

- ☐ Code kompilierbar.
- ☐ Warnung anzeigen.
- ☐ Programm funktioniert.

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.oreilly.de/catalog/heap/steiniger/  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011



## Spielen Sie Compiler, Lösung

Schauen Sie sich das Programm unten an. Ein Teil fehlt. Sie sollten Compiler spielen und sagen, was Sie tun würden, wenn die jeweiligen auf der rechten Seite aufgelisteten Kandidatenfragmente in die Lücke eingesetzt würden.

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



```
printf("Auf dem Merkur hat ein Tag %f Stunden\n", tag);  
return 0;
```

```
}
```

```
float merkurtag_in_erdtagen()
```

```
{
```

```
    return 58.65;
```

```
}
```

```
int stunden_in_erdtag()
```

```
{
```

```
    return 24;
```

```
}
```

```
float merkurtage_in_erdtagen();
int stunden_in_erdtage();

int main()
{
    float taglaenge = merkurtage_in_erdtagen();
    int stunden = stunden_in_erdtage();
    float tag = taglaenge * stunden;
```

- ☒ Code kompilierbar.
- ☐ Warnung anzeigen.
- ☒ Programm funktioniert.

Es gibt eine Warnung, weil stunden\_in\_erdtage() vor dem Aufruf nicht deklariert wurde. Das Programm funktioniert trotzdem, weil der Rückgabebetyp richtig angenommen wird.

```
float merkurtage_in_erdtagen();

int main()
{
    float taglaenge = merkurtage_in_erdtagen();
    int stunden = stunden_in_erdtage();
    float tag = taglaenge * stunden;
```

- ☒ Code kompilierbar.
- ☒ Warnung anzeigen.
- ☒ Programm funktioniert.

```
int main()
{
    float taglaenge = merkurtage_in_erdtagen();
    int stunden = stunden_in_erdtage();
    float tag = taglaenge * stunden;
```

Die Kompilierung scheitert, weil eine float-Funktion vor der Deklaration aufgerufen wird.

- ☐ Code kompilierbar.
- ☒ Warnung anzeigen.
- ☐ Programm funktioniert.

Das Programm kompiliert, funktioniert aber nicht, weil es Rundungsfehler gibt.

```
float merkurtage_in_erdtagen();
int stunden_in_erdtage();

int main()
{
    int taglaenge = merkurtage_in_erdtagen();
    int stunden = stunden_in_erdtage();
    float tag = taglaenge * stunden;
```

Die Variable taglaenge sollte ein float sein.

- ☒ Code kompilierbar.
- ☐ Warnung anzeigen.
- ☐ Programm funktioniert.



## Es gibt keine Dummen Fragen

**F:** `int`-Funktionen muss ich dann also nicht deklarieren?

**A:** Nicht unbedingt, wenn Sie Ihren Code nicht teilen wollen. Mehr dazu erfahren Sie gleich.

**F:** Ich bin verwirrt. Sie erwähnten eine *Vorverarbeitung* durch den Compiler. Warum macht der *Compiler* das?

**A:** Genau genommen kümmert sich der Compiler wirklich nur um den Kompilierungsschritt: Er wandelt den C-Quellcode in Maschinencode um. Allgemeiner gesagt, bezeichnet man sämtliche Phasen der Umwandlung des C-Quellcodes in ein ausführbares Programm als *Kompilierung*, und `gcc` steuert sämtliche Phasen dieses Prozesses – die Vorverarbeitung durch den Präprozessor und die Kompilierung.

**F:** Was ist der Präprozessor?

**A:** Der Präprozessor kümmert sich um die ersten Schritte bei der Verarbeitung des C-Quellcodes in ein funktionierendes Programm. Bei dieser »Vorverarbeitung« wird eine modifizierte Version des Quellcodes erstellt, bevor die *eigentliche* Kompilierung beginnt. In Ihrem Code kümmert sich der Präprozessor darum, dass der Inhalt der Header-Datei in die eigentliche Datei eingelesen wird.

**F:** Erstellt der Präprozessor eine richtige Datei?

**A:** Nein, Compiler nutzen gewöhnlich Pipes, um Daten effizienter zwischen den Phasen des Compilers auszutauschen.

**F:** Warum nutzen manche Header Anführungszeichen, andere hingegen spitze Klammern?

**A:** Genau genommen hängt das von der Funktionsweise Ihres Compilers ab. Anführungszeichen bedeuten üblicherweise, dass über einen relativen Pfad nach einer Datei gesucht wird. Wenn Sie nur den Dateinamen ohne einen Verzeichnisnamen angeben, sucht der Compiler im aktuellen Verzeichnis. Werden spitze Klammern genutzt, sucht er in vorab festgelegten Verzeichnissen.

**F:** In welchen Verzeichnissen sucht der Compiler, wenn er nach Header-Dateien sucht?

**A:** Der `gcc`-Compiler weiß, wo die Standard-Header gespeichert sind. Auf einem Unix-ähnlichen Betriebssystem befinden sich Header-Dateien üblicherweise an Orten wie `/usr/local/include`, `/usr/include` oder ein paar anderen.

**F:** So funktioniert das also bei Standard-Headern wie *stdio.h*?

**A:** Ja. Auf einem Unix-basierten System finden Sie die Datei *stdio.h* unter `/usr/include/stdio.h`. Wenn Sie unter Windows den MinGW-Compiler nutzen, steckt er wahrscheinlich in `C:\MinGW\include\stdio.h`.

**F:** Kann ich eigene Bibliotheken erstellen?

**A:** Ja. Wie Sie das tun, erfahren Sie später in diesem Buch.

## Punkt für Punkt

- Wenn der Compiler auf den Aufruf einer Funktion stößt, von der er noch nicht gehört hat, geht er davon aus, dass die Funktion einen `int` liefert.
- Versuchen Sie, eine Funktion aufzurufen, bevor Sie sie definieren, kann das zu Problemen führen.
- Funktionsdeklarationen sagen dem Compiler, wie Ihre Funktionen aussehen, bevor Sie sie definieren.
- Wenn Funktionsdeklarationen am Anfang Ihres Quellcodes stehen, gerät der Compiler nicht in Verwirrung in Bezug auf den Rückgabebetyp.
- Funktionsdeklarationen werden häufig in Header-Dateien gepackt.
- Sie können dem Compiler mit `#include` sagen, dass er den Inhalt einer Header-Datei lesen soll.
- Der Compiler behandelt durch `#include` eingeschlossenen Code genau so wie Code, der direkt in die Quelldatei eingegeben wurde.



## Diese Tabelle ist reserviert ...

C ist eine sehr kleine Sprache. Hier ist eine vollständige Liste der reservierten Wörter (in keiner bestimmten Abfolge).

Jedes C-Programm, das Ihnen je begegnen wird, wird aus diesen paar Wörtern und ein paar Zeichen bestehen. Wenn Sie eins dieser Wörter als Namen verwenden, wird das den Compiler schrecklich aufregen.

|                      |                       |                       |
|----------------------|-----------------------|-----------------------|
| <code>auto</code>    | <code>if</code>       | <code>break</code>    |
| <code>int</code>     | <code>case</code>     | <code>long</code>     |
| <code>char</code>    | <code>register</code> | <code>continue</code> |
| <code>return</code>  | <code>default</code>  | <code>short</code>    |
| <code>do</code>      | <code>sizeof</code>   | <code>double</code>   |
| <code>static</code>  | <code>else</code>     | <code>struct</code>   |
| <code>entry</code>   | <code>switch</code>   | <code>extern</code>   |
| <code>typedef</code> | <code>float</code>    | <code>union</code>    |
| <code>for</code>     | <code>unsigned</code> | <code>goto</code>     |
| <code>while</code>   | <code>enum</code>     | <code>void</code>     |
| <code>const</code>   | <code>signed</code>   | <code>volatile</code> |

## Bei Gemeinsamkeiten ...

Je mehr C-Programme Sie schreiben, umso wahrscheinlicher wird es, dass Sie feststellen werden, dass Sie Funktionen und Einrichtungen in anderen Programmen wiederverwenden können. Schauen Sie sich beispielsweise die Spezifikationen für die beiden Programme rechts an.

Die XOR-Verschlüsselung ist ein sehr einfaches Verfahren zur Tarnung von Text, bei dem jedes Zeichen per XOR mit einem Wert verknüpft wird. Sie ist nicht sonderlich sicher, lässt sich aber recht einfach durchführen. Und derselbe Code, der zur Verschlüsselung genutzt wurde, kann auch zur Entschlüsselung verwendet werden:

```
void heißt: kein Rückgabewert. → void kodieren(char *nachricht)
{
    char c;
    while (*nachricht) {
        *nachricht = *nachricht ^ 31;
        nachricht++;
    }
}
```

Das Array durchlaufen und alle Zeichen durch eine verschlüsselte Version ersetzen.

Die Funktion erhält einen Zeiger auf ein Array.

Das bedeutet, alle Zeichen per XOR mit der Zahl 31 verknüpfen.

Mit Zeichen rechnen? Das geht, weil char ein numerischer Datentyp ist.

datei\_kodierer

Inhalt einer Datei lesen und eine mit XOR verschlüsselte Version erstellen.

text\_kodierer

Strings von der Standardeingabe lesen und mit XOR verschlüsselte Versionen auf der Standardausgabe ausgeben.

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.oreilly.de/catalog/errata/csp/errata.html  
O'Reilly Verlag 2011

## ... ist teilen gut

Offensichtlich werden beide Programme dieselbe `kodieren()`-Funktion benötigen. Kein Problem, denken Sie? Ich kopiere den Code einfach in das zweite Programm? Wenn die zu kopierende Codemenge klein ist, ist das nicht so aufwendig ... aber was ist, wenn die zu kopierende Codemenge umfangreicher wird? Oder wenn sich die Funktionsweise der Funktion `kodieren()` später ändern muss? Wenn es zwei Exemplare von `kodieren()` gibt, müssen Sie den Code an zwei Stellen ändern.

Möchten Sie dieses Problem auch bei umfangreichem Code im Griff behalten, müssen Sie eine Möglichkeit finden, gemeinsame Codeteile wiederzuverwenden – eine Möglichkeit, Funktionen zu nehmen und sie einem Haufen unterschiedlicher Programme zur Verfügung zu stellen.

## Wie könnte man das tun?

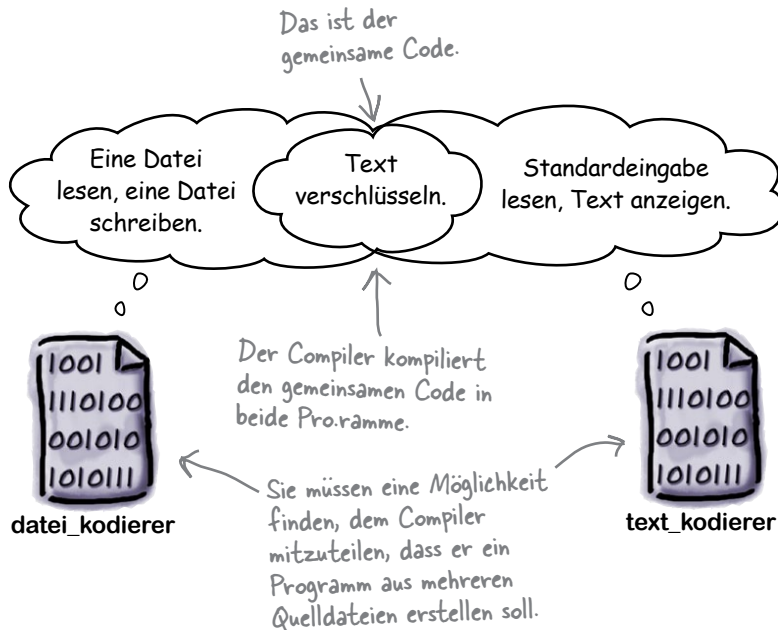


# KOPF- NUSS

Stellen Sie sich vor, Sie möchten eine Gruppe von Funktionen in mehreren Programmen nutzen. Wie hätten Sie das Teilen von Code ermöglicht, wenn Sie der Schöpfer von C gewesen wären?

## Sie können Code auf mehrere Dateien aufteilen

Wenn Sie eine Codemenge haben, die Sie in mehreren Dateien nutzen wollen, ist es sinnvoll, diesen gemeinsam genutzten Code in eine eigene `.c`-Datei zu packen. Sofern der Compiler den gemeinsam genutzten Code irgendwie einschließen kann, wenn er das Programm kompiliert, können Sie den gleichen Code in mehreren Kompilierungseinheiten nutzen. Sollte sich der gemeinsam genutzte Code irgendwann ändern müssen, muss er nur an einer Stelle geändert werden.



Wenn Sie eine eigene `.c`-Datei für den gemeinsamen Code nutzen wollen, haben wir ein *Problem*. Alle Programme, die wir bislang erstellt haben, bestanden nur aus einer einzigen `.c`-Quelldatei. Hätten Sie ein Programm namens `blitz_hack`, hätten Sie es aus einer einzigen Quellcodedatei mit dem Namen `blitz_hack.c` erstellt.

Aber jetzt brauchen Sie eine Möglichkeit, dem Compiler **mehrere Quellcode-dateien zu geben** und dann zu sagen: »Mach mir ein Programm daraus.«

Wie macht man das? Welche Syntax nutzt man dazu beim `gcc`-Compiler? Und wichtiger noch, was heißt es für einen Compiler, eine einzige ausführbare Datei aus mehreren Dateien zu erstellen? Wie würde das funktionieren? Wie würde man sie zusammennähen?

**Wenn wir verstehen wollen, wie der C-Compiler ein Programm aus mehreren Dateien erstellen kann, müssen wir uns ansehen, wie die Kompilierung funktioniert ...**

# Kompilieren – was steckt dahinter?

Wenn wir verstehen wollen, wie der Compiler mehrere Dateien zu einem Programm kompilieren kann, müssen wir den Vorhang lüften und uns ansehen, wie die Kompilierung wirklich funktioniert.

## 1 Präprozessor: Die Quellen reparieren.

Zunächst muss der Compiler die Quellen aufbereiten. Er muss alle Header-Dateien reinmischen, die mit **#include-Direktiven** angegeben wurden. Außerdem muss er eventuell Bereiche des Programms expandieren oder überspringen. Ist das geschehen, ist der Code für die eigentliche Kompilierung bereit.

Das kann er mit Befehlen wie **#define** und **#ifdef** tun. Wie Sie die verwenden, erfahren Sie später im Buch.



Erst mal mische ich den Quellen ein paar weitere Zutaten bei.

»Direktive« ist bloß ein anderes Wort für Befehl.

Ich soll also aus diesen Quelldateien ein Programm zubereiten? Mal sehen, was mir da einfällt ...

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://lpreilly.de/catalog/headers/cger/>  
ung unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

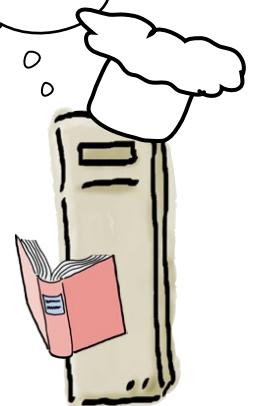
## 2 Kompilierung: In Assembler übersetzen.

Wahrscheinlich erscheint Ihnen C als Programmiersprache schon recht primitiv, trotzdem ist es *immer noch nicht so einfach*, dass der Computer es verstehen kann. Der Computer versteht nur völlig »primitive« Instruktionen in **Maschinensprache**, und der erste Schritt zur Generierung von Maschinensprache ist es, den C-Quellcode in **Assembler-Symbole** folgender Art zu verwandeln:

```
movq -24(%rbp), %rax
movzbl(%rax), %eax
movl %eax, %edx
```

Und? Sieht das primitiv und obskur aus? Assembler beschreibt die einzelnen Instruktionen, die die CPU befolgen muss, wenn sie das Programm ausführt. Der C-Compiler beherrscht einen ganzen Berg an Rezepten für die verschiedenen Teile der Programmiersprache C. Diese Rezepte sagen ihm, wie er eine **if**-Anweisung oder einen Funktionsaufruf in eine Folge von Assembler-Instruktionen übersetzt. Aber für die Maschine ist selbst Assembler nicht einfach genug. Deswegen muss ...

Für diese **if**-Anweisung muss ich zunächst das dem Stack hinzufügen ...



3

### Assembler: Objektcode generieren.

Der Compiler muss die Symbolcodes *assemblieren*, um *Maschinen-* oder **Objektcode** zu erzeugen. Das ist der eigentliche Binärcode, der von den Schaltkreisen in der CPU ausgeführt wird.

Das ist ein  
garstiger Scherz  
in Maschinen-  
sprache.

10010101 00100101 11010101 01011100

Und ist damit alles erledigt? Schließlich wurde der ursprüngliche C-Quellcode in die Nullen und Einsen umgewandelt, die die Schaltkreise des Computers benötigen. Doch nein, es steht immer noch ein Schritt aus. Wenn Sie dem Computer für ein Programm mehrere Dateien zur Kompilierung geben, generiert der Compiler jeweils Objektcode für jede dieser Quelldateien. Aber wenn diese Objektdateien ein ausführbares Programm bilden sollen, muss noch ein Schritt folgen ...

4

### Linken: Die Teile verbinden.

Wenn Sie alle Teile des Objektcodes haben, müssen diese wie die Teile eines Puzzles zusammengesetzt werden, bis sie das vollständige **ausführbare Programm** bilden. Der Compiler verbindet Code in einem Teil des Objektcodes mit Code in einem anderen Teil des Objektcodes, wenn er eine Funktion in jenem Teil aufruft. Das Linken sorgt auch dafür, dass das Programm Bibliothekscode richtig aufrufen kann. Schließlich wird das Programm in eine Programmdatei geschrieben – in einem Format, das vom Betriebssystem unterstützt wird. Das Dateiformat ist wichtig, weil es dem Betriebssystem ermöglicht, das Programm in den Speicher zu laden und auszuführen.



Dies ist ein A  
http://www.o  
Dieser Ausz  
aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
Verlag 2011

**Und wie sagt man dem gcc nun, dass er aus mehreren Dateien eine Programmdatei bauen soll?**

## Der gemeinsame Code braucht einen Header

Wenn Sie den *kodieren.c*-Code in mehreren Programmen nutzen wollen, brauchen Sie eine Möglichkeit, diese Programme über den Verschlüsselungscode zu informieren. Das tun Sie mit einer Header-Datei.

Sie fügen den Header in *kodieren.c* ein.

```
void kodieren(char *nachricht);
```

*kodieren.h*

```
#include "kodieren.h"
```

```
void kodieren(char *nachricht)
{
    char c;
    while (*nachricht) {
        *nachricht = *nachricht ^ 31;
        nachricht++;
    }
}
```

*kodieren.c*

## *kodieren.h* in Ihr Programm einschließen

Hier nutzen Sie die Header-Datei nicht, damit Sie die Funktionen beliebig anordnen können. Sie nutzen sie, um **andere Programme über die *kodieren()*-Funktion zu informieren**:

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

```
int main()
{
    char text[80];
    while (fgets(text, 80, stdin)) {
        kodieren(text);
        printf("%s", text);
    }
}
```

*text\_kodierer.c*

Sie fügen *kodieren.h* ein, damit das Programm die Deklaration der *kodieren()*-Funktion hat.

Wird *kodieren.h* in das Hauptprogramm eingefügt, erfährt der Compiler so viel über die Funktion *kodieren()*, dass er den Code kompilieren kann. Beim Linken kann der Compiler den *kodieren(text)*-Aufruf in *text\_kodierer.c* mit der eigentlichen *kodieren()*-Funktion in *kodieren.h* verbinden.

Um alles gemeinsam zu kompilieren, müssen Sie *gcc* nur noch die Quellcodedateien übergeben

```
gcc text_kodierer.c kodieren.c -o text_kodierer
```

## Variablen teilen

Sie haben gesehen, wie man Funktionen zwischen verschiedenen Dateien teilt. Aber was ist, wenn Sie Variablen teilen wollen? Quellcodedateien haben üblicherweise ihre eigenen Variablen, damit sich eine Variable aus der einen Datei nicht auf eine Variable gleichen Namens in einer anderen Datei auswirken kann. Wollen Sie jedoch unbedingt Variablen teilen, deklarieren Sie diese in der Header-Datei und stellen ihr das Schlüsselwort **extern** voran:

```
extern int passcode;
```

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.oreilly.de/catalog/errata/csp/errata.html  
Dieser Auszug ist  
© 2011





# TESTLAUF

Schauen wir uns an, was passiert, wenn Sie das `text_kodierer`-Programm kompilieren:

Wenn Sie das Programm ausführen, können Sie Text eingeben und sich die verschlüsselte Version ansehen.

Sie können ihm sogar den Inhalt der Datei `kodieren.h` zur Verschlüsselung übergeben.

Das `text_kodierer`-Programm nutzt die `kodieren()`-Funktion aus `kodieren.c`.

```
Datei Bearbeiten Fenster Hilfe Shhh...
> gcc text_kodierer.c kodieren.c -o text_kodierer
> ./text_kodierer
Das ist eine geheime Botschaft!
[~1?vlk?zvqz?xzwzvz?]pk1|w~yk>
> ./text_kodierer < kodieren.h
ipv{?tp{vzmzq7|w~m?5q~|wmv|wk6$
>
```

Sie müssen den Code mit beiden Dateien kompilieren.

Das Programm funktioniert. Nachdem Sie die `kodieren()`-Funktion in eine separate Datei gesteckt haben, können Sie sie in jedem beliebigen Programm nutzen. Wenn Sie `kodieren()` je ändern wollen, beispielsweise um die Funktion etwas sicherer zu machen, müssen Sie nur an der Datei `kodieren.c` arbeiten.

## Punkt für Punkt

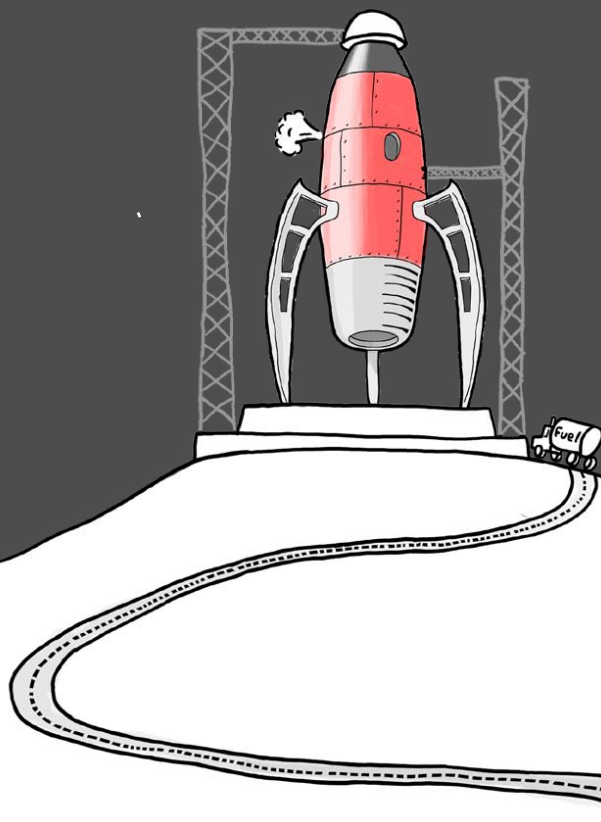
- Sie können Code teilen, indem Sie ihn in eine eigene C-Datei stecken.
- Sie müssen die Funktionsdeklaration in eine eigene `.h`-Header-Datei stecken.
- Schließen Sie die Header-Datei in alle C-Dateien ein, die den gemeinsamen Code nutzen müssen.
- Führen Sie alle benötigten C-Dateien im Kompilierungsbefehl auf.



## Abseits der Piste

Schreiben Sie ein eigenes C-Programm, das die Funktion `kodieren()` nutzt. Denken Sie daran, dass Sie die gleiche Funktion für die Entschlüsselung des Texts nutzen können.

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.oreilly.de/catalog/headsinger/  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011



Mensch! Jedes Mal, wenn ich eine kleine Änderung an **einer** Datei vornehme, dauert es eine Ewigkeit, bis das Programm neu kompiliert ist! Ist ja nicht so, als hätte ich alle Zeit der Welt ...

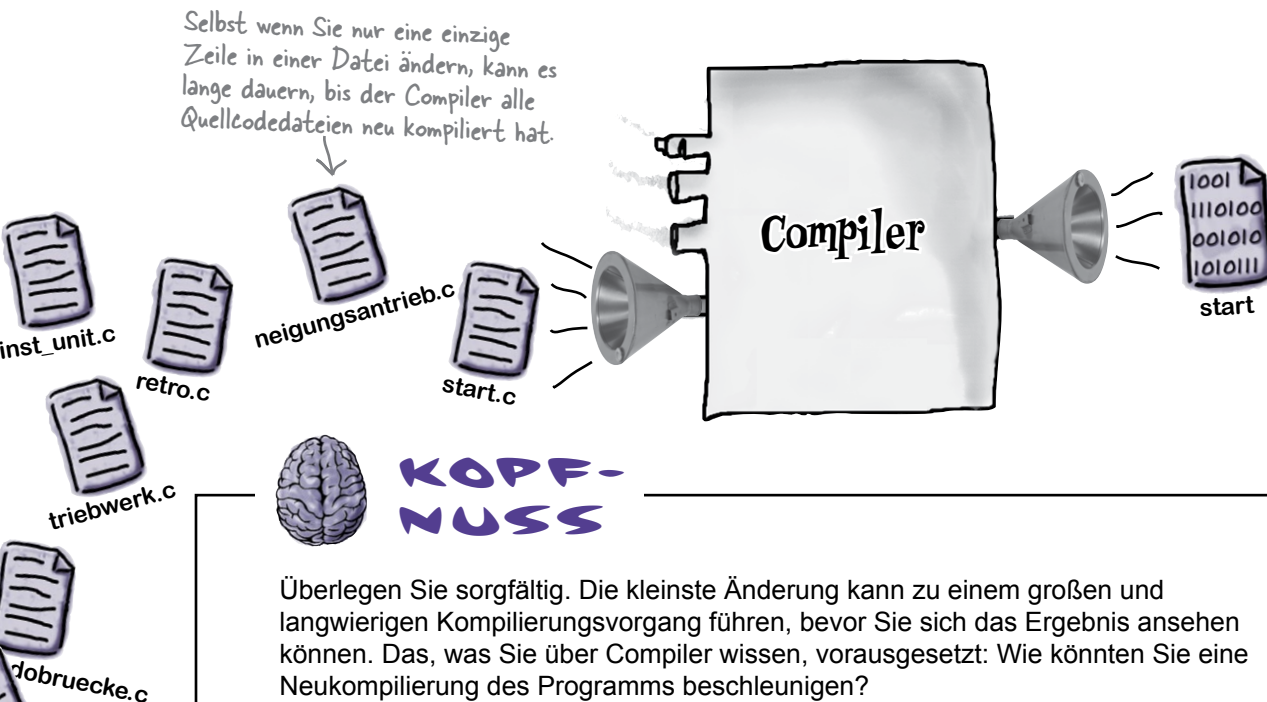


## Es ist kein Mysterium ... oder doch?

Dass Sie Ihr Programm auf mehrere Quelldateien verteilen, sorgt nicht nur dafür, dass Sie *den gleichen Code* in mehreren Programmen nutzen können, es heißt auch, dass Sie damit beginnen können, so *richtig große* Programme aufzubauen. Warum das? Weil Sie jetzt damit Ihr Programm in kleinere, **in sich abgeschlossene** Codeeinheiten gliedern können. Sie sind nicht gezwungen, mit einer *gewaltigen* Quelldatei zu arbeiten, sondern können stattdessen viele *einfachere* Dateien nutzen, die leichter zu verstehen, zu warten und zu testen sind.

Auf der Habenseite sehen Sie also, dass Sie jetzt so richtig große Programme aufbauen können. Und auf der Sollseite? Auch dort steht, dass Sie nun beginnen können, richtig große Programme aufzubauen! C-Compiler sind ausgesprochen effiziente Softwaregeschöpfe. Sie unterziehen Ihr Programm einigen sehr komplexen Transformationen, sie können Ihren Quelltext modifizieren, Hunderte von Dateien zusammenbinden, ohne dass der Speicher hochgeht, und nebenbei auch noch den Code optimieren, den Sie geschrieben haben. Und obwohl sie all diese Dinge tun, schaffen sie es trotzdem, schnell zu laufen.

Aber wenn Sie Programme erstellen, die mehr als ein paar Dateien umfassen, wird die Zeit, die die Kompilierung des Codes in Anspruch nimmt, relevant. Angenommen, es dauert eine Minute, um ein großes Projekt zu kompilieren. Das hört sich gar nicht so viel an, aber es ist mehr als genug, um Ihren Arbeitsfluss zu unterbrechen. Wenn Sie eine Änderung an einer Codezeile vornehmen, wollen Sie die Folgen dieser Änderung so schnell wie möglich sehen. Wenn Sie jedes Mal eine vollständige Minute warten müssen, bis Sie sich die Auswirkungen Ihrer Änderungen ansehen können, wird Sie das erheblich ausbremsen.



Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/headers/cger/>  
 Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

## Nicht alles neu kompilieren

Wenn Sie nur eine oder zwei Quellcodedateien geändert haben, ist es Verschwendung, alle Quelldateien für Ihr Programm neu zu kompilieren. Überlegen Sie, was passiert, wenn Sie einen Befehl folgender Art absetzen:

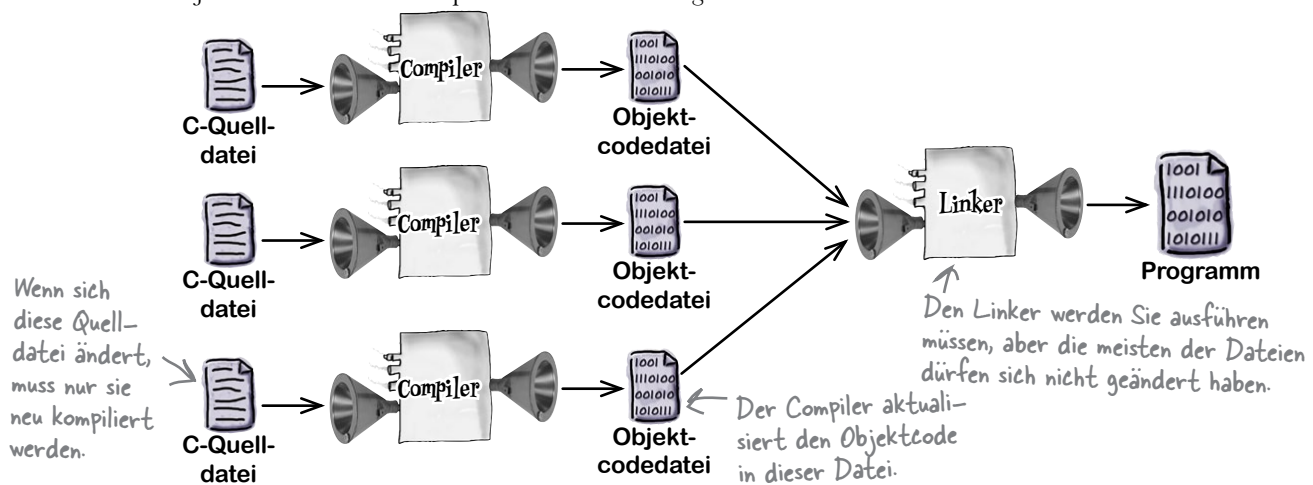
```
gcc rcs.c neigungsantrieb.c ... triebwerk.c -o start
```

Hier werden ein paar Dateinamen weggelassen.

Was tut der Compiler jetzt? Er führt die Präprozessor-, Compiler- und Assembler-Schritte für *jede Quelldatei* durch. Auch für jene, die sich nicht geändert haben. Aber wenn sich der Quellcode nicht geändert hat, ändert sich auch der für diese Datei generierte **Objektcode** nicht. Was also müssen Sie tun, wenn der Compiler jedes Mal für alle Dateien Objektcode generiert?

## Kopien des generierten Codes speichern

Wenn Sie dem Compiler sagen, dass er den generierten Objektcode in einer Datei speichern soll, müsste er ihn eigentlich nur dann neu erzeugen, wenn sich die Quelldatei ändert. Sobald sich eine Datei *ändert*, können Sie den Objektcode für diese **eine Datei** neu erstellen lassen und dann den ganzen Satz von Objektdateien an den Compiler zum Linken übergeben.



Wenn Sie nur eine Datei ändern, müssen Sie die Objektcodedatei für diese Datei neu erstellen, aber für keine der anderen Dateien. Anschließend übergeben Sie sämtliche Objektcodedateien an den Linker und erstellen eine neue Version des Programms.

**Aber wie sagen Sie gcc, dass er den Objektcode in einer Datei speichern soll? Und wie bringen Sie den Compiler dazu, die Objektdateien zu linkern?**



# Erst aus Quellen Objektdateien machen

Sie brauchen Objektcode für alle Quelldateien, und das erreichen Sie mit folgendem Befehl:

Das erstellt den Objektcode für die Datei. `gcc -c *.c` ← Das Betriebssystem ersetzt \*.c durch alle C-Dateinamen.

Das \*.c wird zu allen C-Dateien im aktuellen Verzeichnis expandiert, und das -c sagt dem Compiler, dass für jede C-Datei eine Objektdatei erstellt werden soll, dass diese aber noch nicht zur ausführbaren Programmdatei verlinkt werden sollen.

## Dann Objektdateien linken

Wenn Sie einen Satz von Objektdateien haben, können Sie diese mit einem einfachen Kompilierungsbehl linkn. Aber dabei geben Sie dem Compiler nicht die Namen der C-Quelldateien, sondern die Namen der Objektdateien:

Das ähnelt den Compiler-Befehlen, die Sie bereits genutzt haben. `gcc *.o -o start` ← Führen Sie statt der C-Quelldateien die Objektdateien auf.  
Das findet alle Objektdateien im aktuellen Verzeichnis.

Der Compiler ist clever genug, Objektdateien als Objektdateien und nicht als Quelldateien zu erkennen, überspringt also die meisten Kompilierungsschritte und verlinkt bloß noch die Objektdateien zu einem ausführbaren Programm namens start.

Schön, jetzt haben Sie das Programm kompiliert wie vorher auch. Aber Sie haben auch einen Satz von Objektdateien, die zur Verlinkung bereit sind, sobald Sie sie wieder benötigen. Wenn Sie eine der Dateien ändern, müssen Sie also nur diese neu kompilieren und können das Programm dann wieder linkn:

Nur diese Datei hat sich geändert. `gcc -c steuerduese.c` ← Das erstellt die Datei steuerduese.o neu.  
`gcc *.o -o start` ← Das linkt alles zusammen.

Obleich Sie jetzt zwei Befehle eingeben müssen, sparen Sie eine Menge Zeit:

|                | vorher       | nachher |
|----------------|--------------|---------|
| Kompilierzeit: | 2 min 30 sek | 2 sek   |
| Linkzeit:      | 6 sek        | 6 sek   |

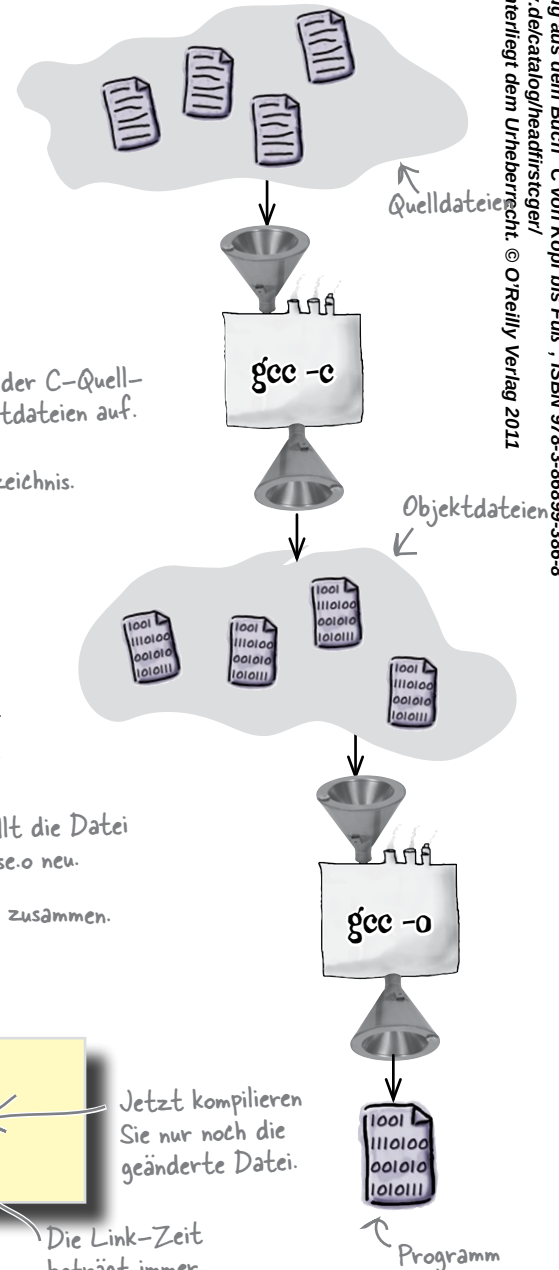
Vorher haben Sie alle Dateien kompiliert.

Die Kompilierung ist um 95 % schneller.

Die Link-Zeit beträgt immer noch 6 Sekunden.

Jetzt kompilieren Sie nur noch die geänderte Datei.

**gcc -c kompiliert den Code, linkt ihn aber nicht.**

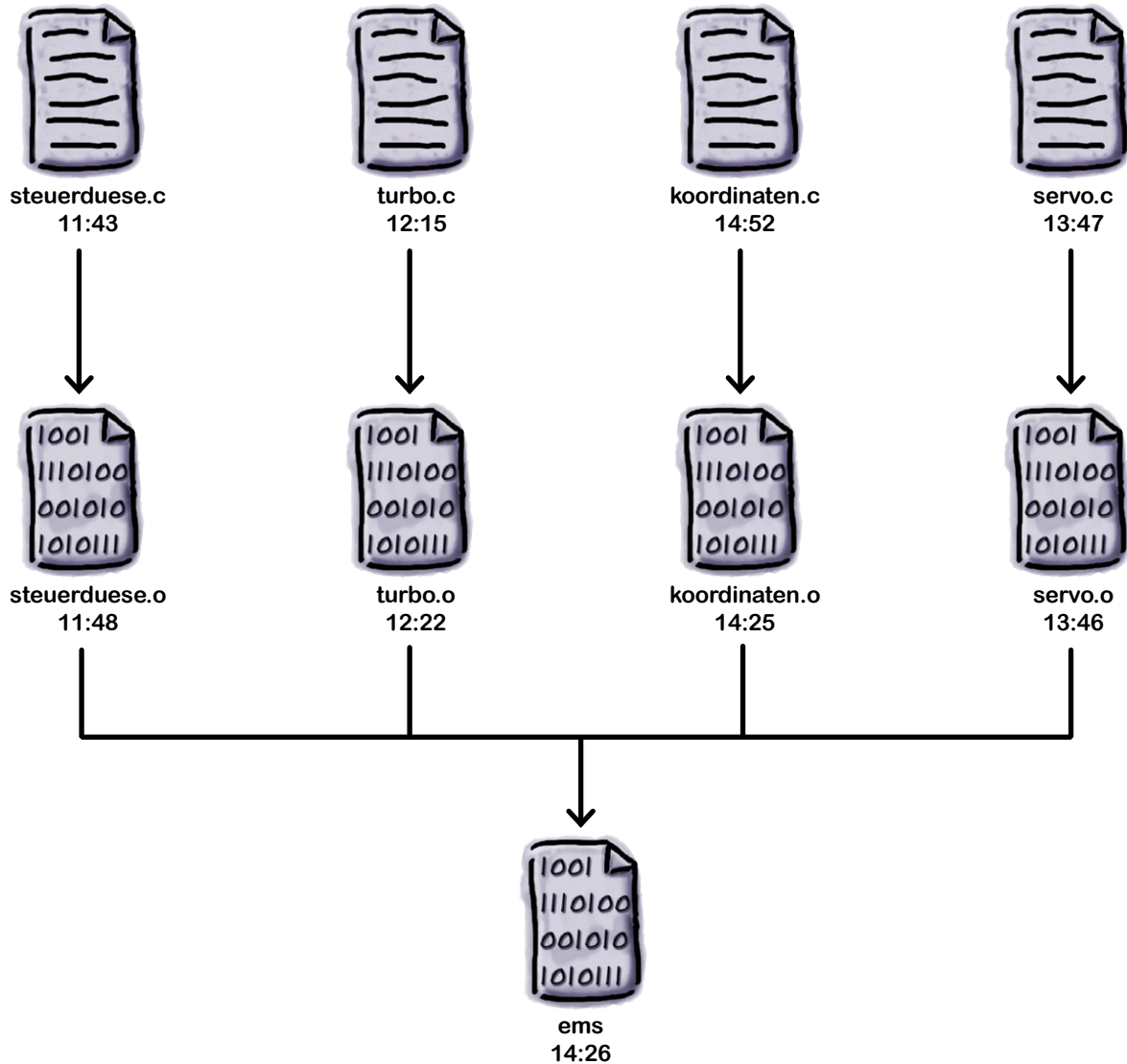


Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.oreilly.de/catalog/headsinger/  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011



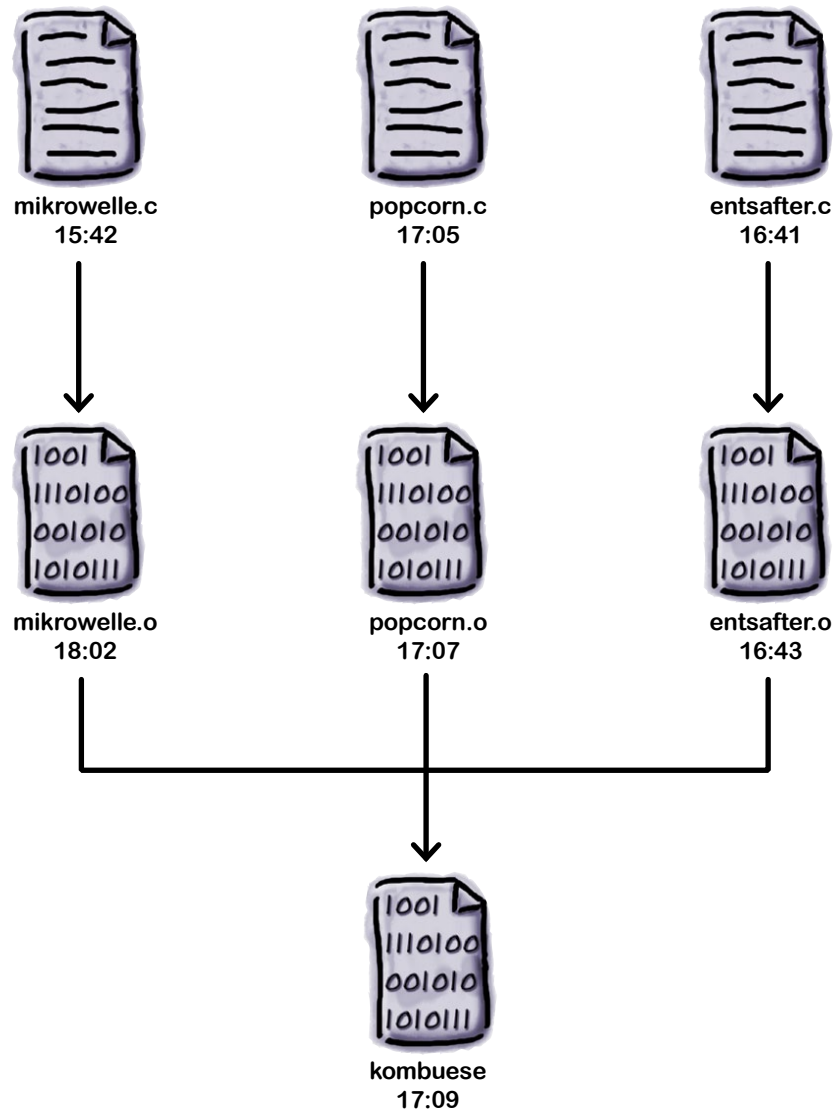
## Lange Übung

Hier ist etwas Code, der zur Steuerung der Schiffsturbinen eingesetzt wird. Alle Dateien tragen einen Zeitstempel. Welche Dateien müssen neu erstellt werden, damit das Programm `ems` wieder aktuell ist? Kreisen Sie die Dateien ein, die aktualisiert werden müssen.



Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/headsrtscger/>  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Und in der Kombüse muss auch geprüft werden, ob der Code aktuell ist. Schauen Sie sich wieder die Zeitstempel der Dateien an. Welche Dateien müssen aktualisiert werden?



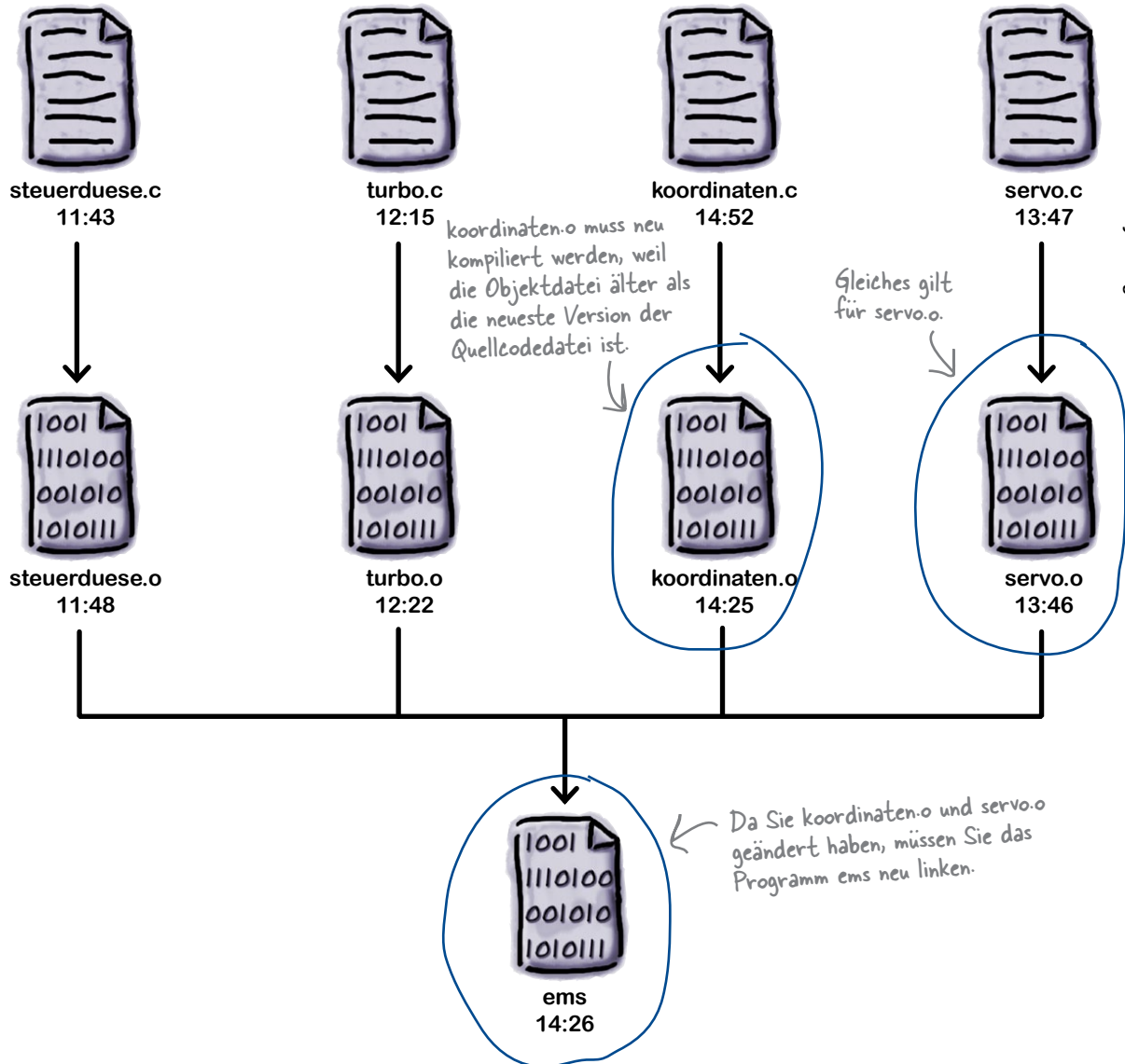
Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/heads/rscger/>  
 Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011





## lange Übung Lösung

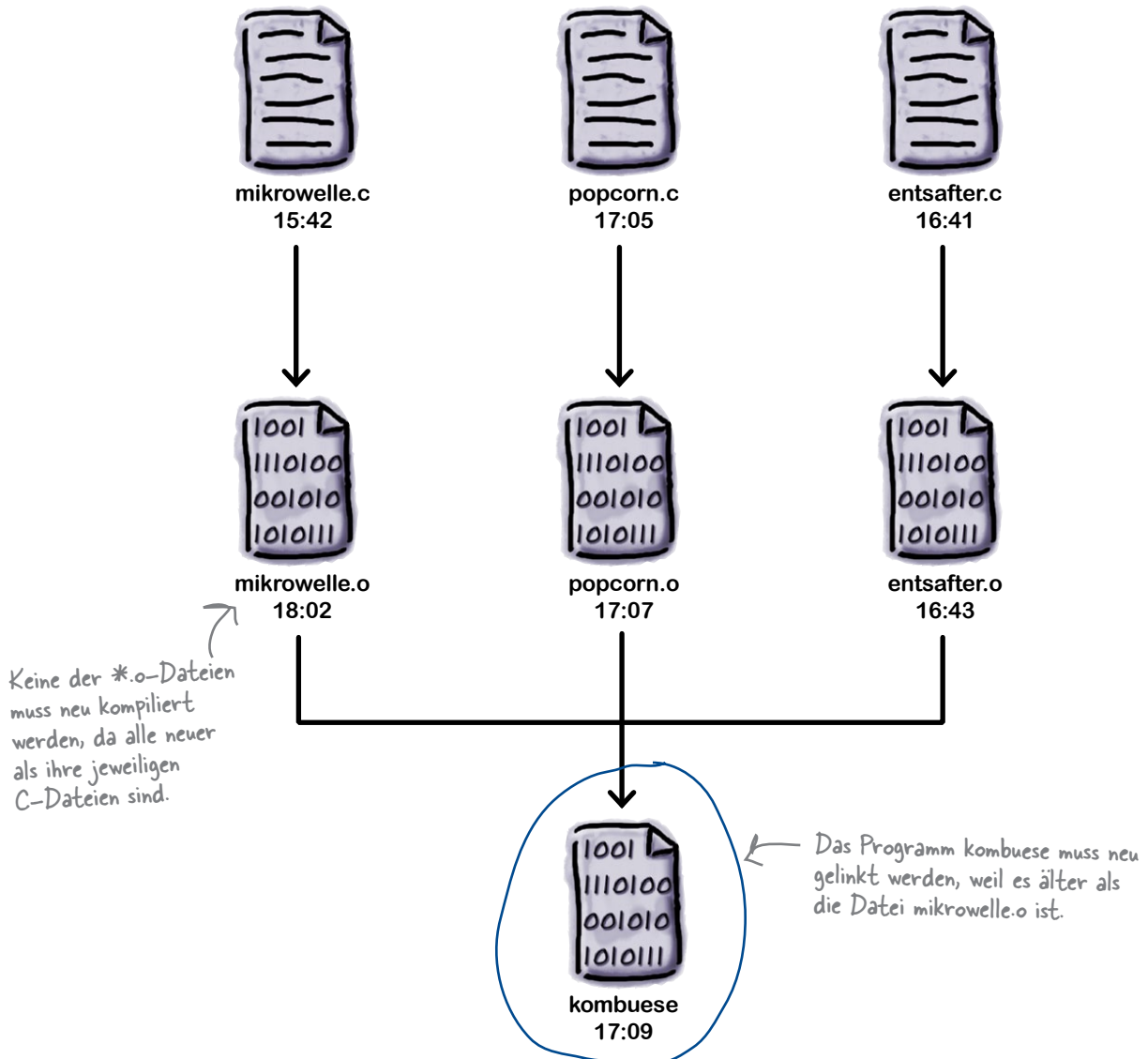
Hier ist etwas Code, der zur Steuerung der Schiffsturbinen eingesetzt wird. Alle Dateien tragen einen Zeitstempel. Welche Dateien müssen neu erstellt werden, damit das Programm `ems` wieder aktuell ist? Sie sollten die Dateien einkreisen, die aktualisiert werden müssen.



Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/headsrtscger/>  
 Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

Und in der Kombüse muss auch geprüft werden, ob der Code aktuell ist. Sie sollten ermitteln, welche Dateien aktualisiert werden mussten.

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/headsrtscger/>  
 Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011



## Das Nachhalten der Dateien ist aufwendig



Ich **denke**, der Sinn des Zeitsparens läge darin, dass wir nicht so stark abgelenkt werden. Jetzt läuft zwar die Kompilierung schneller, aber ich muss darüber nachdenken, wie ich meinen Code kompiliere. Das bringt doch nichts.

**Das stimmt: Teilkompilierungen sind schneller, aber Sie müssen sich mehr Gedanken darüber machen, dass Sie tatsächlich die richtigen Dinge kompilieren.**

Solange Sie nur an einer einzigen Quellcodedatei arbeiten, wird die Sache recht klar sein. Aber wenn Sie mehrere Dateien geändert haben, vergisst man sehr schnell, einige davon neu zu kompilieren. Das führt dazu, dass in das neu kompilierte Programm nicht alle Änderungen übernommen werden, die Sie vorgenommen haben. Wenn Sie Ihr Programm endgültig *ausliefern*, können Sie natürlich darauf achten, dass eine vollständige Neukompilierung *aller* Dateien erfolgt, aber während Sie den Code entwickeln, wollen Sie das tunlichst vermeiden.

Obgleich die Suche nach den Dateien, die kompiliert werden müssen, ein recht **mechanischer Vorgang** ist, werden Sie sicher schnell eine Änderung übersehen, wenn Sie das manuell tun.

Gibt es nicht etwas, das wir nutzen können, um den **Vorgang zu automatisieren?**



Wäre es nicht wunderbar, wenn es ein  
Werkzeug gäbe, das automatisch nur das  
neu kompiliert, was sich geändert hat?  
Aber das ist sicher nur ein Traum ...

## Die Erstellung mit make automatisieren

Sie können Ihre Anwendungen mit `gcc` recht schnell kompilieren, solange Sie sich merken, welche Dateien sich geändert haben. Manuell wäre das eine recht komplizierte Sache, die sich glücklicherweise aber leicht automatisieren lässt. Stellen Sie sich vor, Sie haben eine Datei, die auf Basis einer anderen Datei erstellt wurde. Nehmen wir an, es sei eine Objektdaten, die aus einer Quelldatei kompiliert wird:

Wenn `steuerduese.c` neuer ist, müssen Sie neu kompilieren. → `steuerduese.c` → `steuerduese.o` ← Wenn die Datei `steuerduese.o` neuer ist, müssen Sie nicht neu kompilieren.

Wie stellen Sie fest, ob die Datei `steuerduese.o` neu kompiliert werden muss? Sie schauen sich einfach die Zeitstempel der beiden Dateien an. Wenn die Datei `steuerduese.o` älter als die Datei `steuerduese.c` ist, muss die Datei `steuerduese.o` neu erstellt werden. Andernfalls ist sie aktuell.

Das ist eine sehr einfache Regel. Und wenn Sie für irgendetwas eine einfache Regel haben, denken Sie nicht weiter darüber nach – **automatisieren Sie sie** ...

**make** ist ein Werkzeug, das für Sie den Kompilierungsbefehl ausführen kann. `make` prüft für Sie den Zeitstempel der Quelldateien und der generierten Dateien und kompiliert die Dateien nur, wenn etwas nicht mehr aktuell ist.

Aber bevor Sie all das machen können, müssen Sie `make` über Ihren Quellcode informieren. Es muss genau wissen, welche Dateien von welchen Dateien abhängen. Und Sie müssen ihm auch genau sagen, wie der Code erstellt werden soll.

### Was muss make wissen?

Die Dateien, die `make` kompiliert, bezeichnet man als **Target** oder **Ziel**. Genau genommen ist `make` nicht auf das Kompilieren von Dateien beschränkt. Ein Ziel ist eine beliebige Datei, die auf Basis von anderen Dateien *generiert* wird. Das Ziel könnte also ebenso ein Zip-Archiv sein, das auf Basis von zu komprimierenden Dateien erstellt wird.

Für jedes Ziel muss `make` *zwei Dinge* wissen:



#### Die Abhängigkeiten.

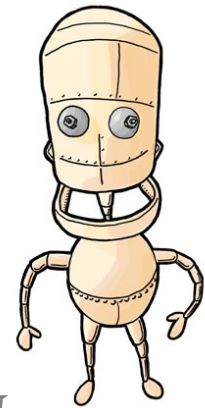
Die Dateien, auf deren Basis das Ziel generiert wird.



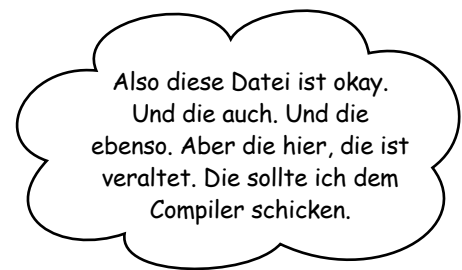
#### Das Rezept.

Der Satz an Anweisungen, die ausgeführt werden müssen, um die Datei zu generieren.

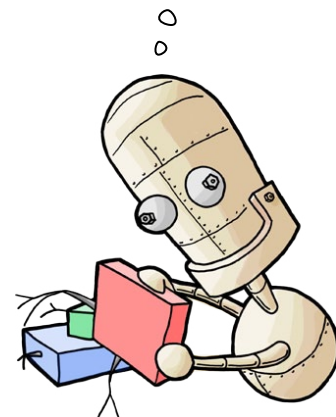
Gemeinsam bilden die Abhängigkeiten und das Rezept eine **Regel**. Eine Regel sagt `make` alles, was es wissen muss, um die Zieldatei zu erstellen.



Das ist `make`, Ihr neuer bester Freund.



Also diese Datei ist okay. Und die auch. Und die ebenso. Aber die hier, die ist veraltet. Die sollte ich dem Compiler schicken.



# Wie make funktioniert

Angenommen, Sie wollen *steuerduese.c* zu Objektcode in *steuerduese.o* kompilieren. Was sind die Abhängigkeiten, und was ist das Rezept?

`steuerduese.c` → `steuerduese.o`

Die Dateien *steuerduese.o* ist das **Ziel**, weil das die Datei ist, die Sie generieren wollen. *steuerduese.c* ist eine Abhängigkeit, weil das eine Datei ist, die der Compiler braucht, um *steuerduese.o* zu erstellen. Und was ist das Rezept? Das ist der Kompilierungsbehehl, der *steuerduese.c* in *steuerduese.o* umwandelt.

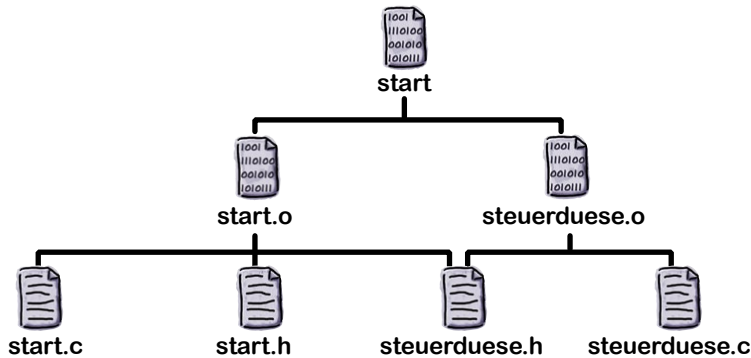
`gcc -c steuerduese.c` ← Das ist die Regel zur Erstellung von *steuerduese.o*.

Verstanden? Wenn Sie *make* über die Abhängigkeiten und das Rezept informieren, können Sie *make* die Entscheidung überlassen, wann *steuerduese.o* neu kompiliert werden muss.

Aber Sie können noch weitergehen. Wenn Sie die Datei *steuerduese.o* erstellen, können Sie sie nutzen, um das Programm *start* zu erstellen. Das heißt, dass die Datei *start* auch als Ziel eingerichtet werden kann, weil das eine Datei ist, die Sie generieren wollen. Die Abhängigkeiten für *start* sind alle *.o*-Objektdateien. Das Rezept ist dieser Befehl:

`gcc *.o -o start`

Kennt *make* alle Details der Abhängigkeiten und Regeln, müssen Sie ihm nur sagen, dass es die Datei *start* erstellen soll. *make* kümmert sich dann um die Details.



**Aber wie informieren Sie *make* über die Abhängigkeiten und Rezepte? Finden wir es heraus.**

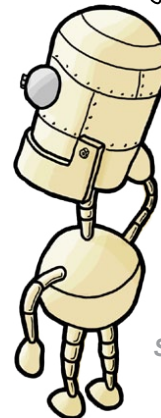


**make kann unter Windows einen anderen Namen haben.**

Weil *make* aus der Unix-Welt stammt, gibt es unter Windows verschiedene Varianten. MinGW enthält eine *make*-Version namens *mingw32-make*, und Microsoft bietet eine eigene Version namens *NMAKE*.

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
http://www.oreilly.de/catalog/headsinger/  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

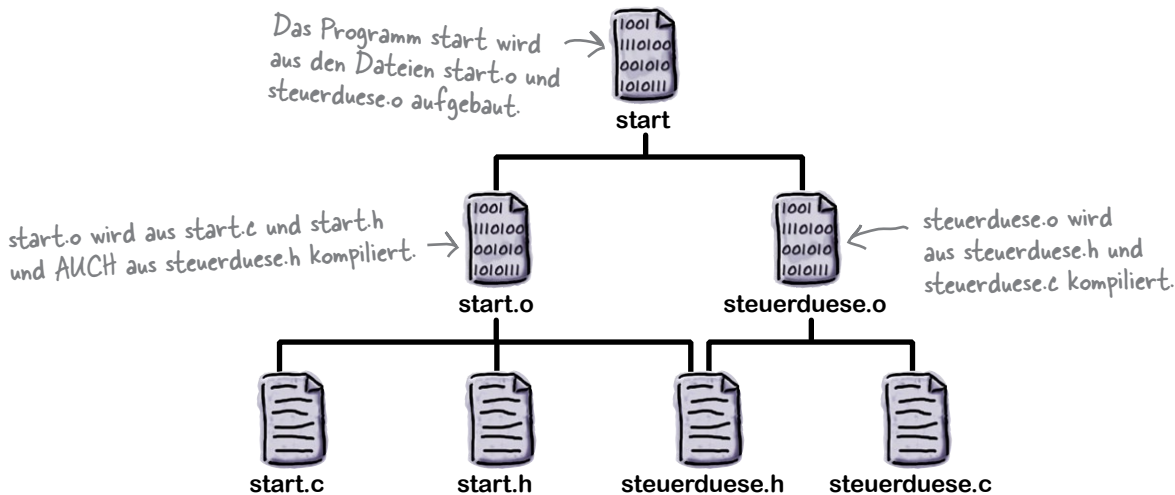
Ich soll also das Programm *start* kompilieren? Dann muss ich zuerst *steuerduese.o* neu kompilieren, weil es veraltet ist. Und dann brauche ich *start* nur noch neu zu linken.





# make mit einem makefile über Ihren Code informieren

Alle Informationen über Targets, Abhängigkeiten und Rezepte müssen in einer Datei gespeichert werden, die entweder *makefile* oder *Makefile* heißt. Wenn Sie sich vorstellen wollen, wie das funktioniert, stellen Sie sich vor, Sie hätten einen Satz von Quelldateien, die gemeinsam das *start*-Programm bilden:



Das Programm *start* wird erstellt, indem die Dateien *start.o* und *steuerduese.o* verlinkt werden. Diese Dateien werden aus den entsprechenden C-Dateien und Header-Dateien kompiliert, aber die Datei *start.o* hängt auch von der Datei *steuerduese.h* ab, weil sie Code enthält, der eine Funktion im *steuerduese*-Code aufrufen muss.

So beschreiben Sie diesen Erstellungsvorgang in einem Makefile:



**Alle Rezeptzeilen MÜSSEN mit einem Tabulatorzeichen beginnen.**

Wenn Sie versuchen, die Zeilen mit Leerzeichen einzurücken, funktioniert das Erstellen nicht.





# TESTLAUF

Speichern Sie Ihre make-Regeln in einer Textdatei mit dem Namen *Makefile* im gleichen Verzeichnis. Öffnen Sie dann die Konsole und geben Sie Folgendes ein:

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/hoedlrichs/crnr/>  
 Dieser Auszug unterli

1

Sie sagen make, dass es die Datei start erstellen soll.  
 Erst erstellt make mit dieser Zeile start.o.  
 Dann muss make mit dieser Zeile steuerduese.o erstellen.  
 Schließlich linkt make die Objektdatei, um das Programm start zu erstellen.

```
Datei Bearbeiten Fenster Hilfe MachEsSo
> make start
gcc -c start.c
gcc -c steuerduese.c
gcc start.o steuerduese.o -o start
```

Sie sehen, dass make die Abfolge der Befehle herausfinden konnte, die zur Erstellung von start erforderlich ist. Aber was ist, wenn Sie eine Änderung an der Datei *steuerduese.c* vornehmen und dann wieder make ausführen?

make muss start.c nicht mehr kompilieren.

start.o ist bereits aktuell.

```
Datei Bearbeiten Fenster Hilfe MachEsSo
> make start
gcc -c steuerduese.c
gcc start.o steuerduese.o -o start
```

make kann die Erstellung einer neuen Version von *start.o* überspringen. Stattdessen wird nur *steuerduese.o* kompiliert, und dann wird das Programm neu gelinkt.

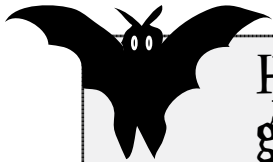
## Es gibt keine Dummen Fragen

**F:** Ist `make` wie `ant`?

**A:** Wahrscheinlich vergleicht man Build-Werkzeuge wie `ant` und `rake` besser mit `make`. `make` war eins der ersten Werkzeuge, die zur Automatisierung der Erstellung von Programmen aus Quellcode entwickelt wurden.

**F:** Es scheint eine Menge Arbeit zu sein, nur um den Quellcode zu kompilieren. Bringt das wirklich etwas?

**A:** Ja, wirklich, `make` ist ungewöhnlich nützlich. Bei kleinen Projekten scheint `make` vielleicht nicht so viel Zeit zu sparen, aber wenn man erst einmal eine Handvoll Dateien hat, kann das Kompilieren und Linken eine ziemliche Qual werden.



### Horror- geschichten

#### Warum mit Tabulatoren einrücken?

*Man könnte die Rezepte ebenso gut mit Leerzeichen wie mit Tabulatoren einrücken. Warum besteht `make` auf dem Einsatz von Tabulatoren? Hier ist ein Zitat vom Schöpfer von `make`, Stuart Feldman:*

*»Wieso der Tabulator in Spalte 1? ... Er funktionierte, also blieb er. Und nach ein paar Wochen hatte ich dann rund ein Dutzend Nutzer, die meisten davon Freunde, und ich wollte meiner Gefolgschaft nicht schaden. Der Rest ist, traurigerweise, Geschichte.«*

**F:** Wenn ich ein Makefile für ein Windows-System schreibe, funktioniert das dann auch auf einem Mac oder einem Linux-Rechner?

**A:** Weil Makefiles Befehle auf dem zugrunde liegenden Betriebssystem ausführen, funktionieren sie auf anderen Betriebssystemen gelegentlich nicht.

**F:** Kann ich Makefiles zu anderen Dingen als zur Kompilierung von Code verwenden?

**A:** Ja, `make` wird am häufigsten zur Kompilierung von Code eingesetzt. Aber es kann auch als Kommandozeilen-Installationsprogramm oder als Werkzeug für die Versionskontrolle dienen. Im Prinzip können Sie es für alle Aufgaben einsetzen, die Sie auf der Kommandozeile ausführen können.



### Freak-Futter

`make` nimmt der Kompilierung von Dateien eine Menge des Schreckens. Aber sollten Sie den Eindruck haben, dass das nicht reicht, sollten Sie sich ein Werkzeug namens **autoconf** ansehen.

<http://www.gnu.org/software/autoconf/>

`autoconf` wird zur Generierung von Makefiles genutzt. C-Programmierer erstellen häufig Werkzeuge zur Automatisierung der Erstellung von Software. Eine wachsende Zahl von ihnen ist auf der GNU-Website verfügbar.

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/heads/cgcr/>  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011



## Make-Magneten

Finden Sie auch, dass das, was man heute so an Musik vorgesetzt bekommt, recht schwungfrei ist? Dann wird Ihnen das Programm gefallen, das die Jungs aus der KopfüBar geschrieben haben! `oggswing` ist ein Programm, das eine Ogg Vorbis-Audiodatei einliest und eine Swingversion davon erstellt. Nice! Schauen Sie, ob Sie das `makefile` vervollständigen können, das `oggswing` kompiliert und es dann nutzt, um eine `.ogg`-Datei zu transformieren:

Wandelt  
`whitennerdy.ogg`  
in `swing.ogg` um.

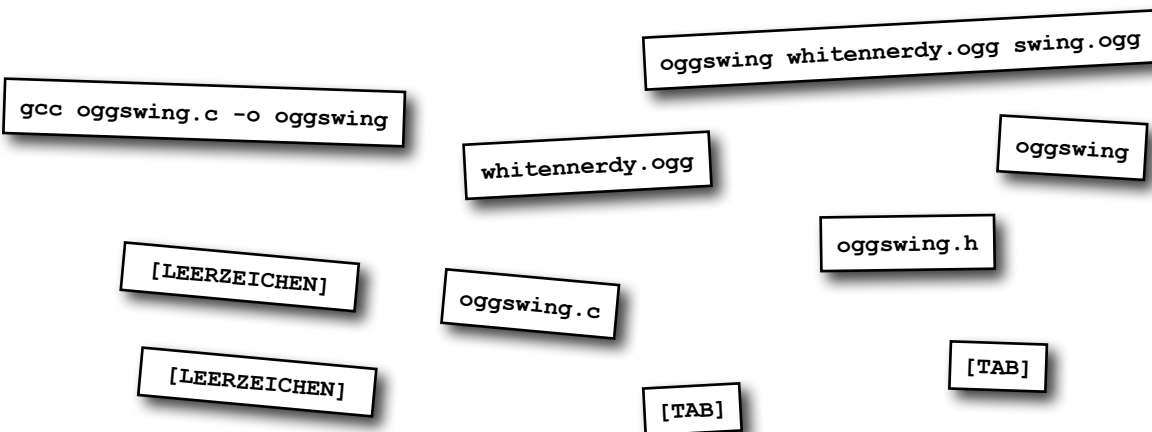


`oggswing:` .....

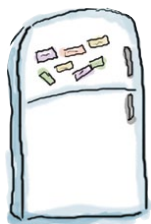
.....

`swing.ogg:` .....

.....



Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/headsfirstcger/>  
Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011



## Make-Magneten, Lösung

Finden Sie auch, dass das, was man heute so an Musik vorgesetzt bekommt, recht schwungfrei ist? Dann wird Ihnen das Programm gefallen, das die Jungs aus der KopfüBar geschrieben haben! `oggswing` ist ein Programm, das eine Ogg Vorbis-Audiodatei einliest und eine Swingversion davon erstellt. Nice! Sie sollten das `makefile` vervollständigen, das `oggswing` kompiliert und es dann nutzt, um eine `.ogg`-Datei zu kompilieren:

```
oggswing: oggswing.c oggswing.h
[TAB] gcc oggswing.c -o oggswing

swing.ogg: whitennerdy.ogg oggswing
[TAB] oggswing whitennerdy.ogg swing.ogg
```



## Freak-Futter

[LEERZEICHEN]

[LEERZEICHEN]

Das `make`-Werkzeug kann erheblich mehr, als wir hier betrachten können. Mehr über `make` und das, was es für Sie leisten kann, erfahren Sie im *GNU Make Manual* unter:

<http://tinyurl.com/yczmjx>

Dies ist ein Auszug aus dem Buch "C von Kopf bis Fuß", ISBN 978-3-86899-386-8  
<http://www.oreilly.de/catalog/headers/cger/>  
 Dieser Auszug unterliegt dem Urheberrecht. © O'Reilly Verlag 2011

# Abheben!

Wenn Ihr Erstellungsprozess sehr langsam läuft, kann `make` ihn erheblich beschleunigen. Die meisten Entwickler sind so daran gewöhnt, ihren Code mit `make` zu erstellen, dass sie es sogar für kleine Programme verwenden. `make` ist, als hätten Sie einen sehr sorgfältigen Entwickler an Ihrer Seite. Wenn Sie viel Code schreiben müssen, sorgt `make` dafür, dass jeweils nur der Code kompiliert wird, der zur jeweiligen Zeit benötigt wird.

**Und manchmal ist es einfach wichtig, dass alles zeitig erledigt wird ...**

## Punkt für Punkt

- Es kann eine lange Zeit dauern, eine große Menge Dateien zu kompilieren.
- Sie können den Kompilierungsvorgang beschleunigen, indem Sie Objektcode in \*.o-Dateien speichern.
- `gcc` kann Programme aus Objektdateien ebenso kompilieren wie aus Quelldateien.
- Das `make`-Werkzeug kann Erstellungsprozesse automatisieren.
- `make` kennt die Abhängigkeiten zwischen Dateien und kann deswegen nur die Dateien kompilieren, die sich geändert haben.
- `make` muss über den Erstellungsvorgang mit einem Makefile informiert werden.
- Achten Sie in Makefiles auf die Formatierung: Vergessen Sie nicht, dass Sie für den Einzug Tabulatoren statt Leerzeichen nutzen müssen.



## Ihr C-Werkzeugkasten

**Jetzt haben Sie Kapitel 4 intus und Ihrem Werkzeugkasten Datentypen, Header-Dateien und make einverleibt. Eine vollständige Liste aller Werkzeuge in diesem Buch finden Sie in Anhang B.**

chars sind  
Zahlen.

Nutzen Sie  
shorts für  
kleine ganze  
Zahlen.

Nutzen Sie ints  
für die meisten  
ganzen Zahlen.

Nutzen Sie  
longs für  
große ganze  
Zahlen.

Nutzen Sie  
floats für die  
meisten Gleit-  
kommazahlen.

Nutzen Sie  
doubles für  
richtig genaue  
Gleitkomma-  
zahlen.

Trennen Sie  
Funktionsde-  
klarationen von  
-definitionen.

Stecken Sie  
Deklarationen  
in eine Hea-  
der-Datei.

#include  
<> für  
Bibliotheks-  
Header.

#include ""  
für lokale  
Header.

Speichern Sie  
Objektcode in  
Dateien, um  
die Erstellung  
zu beschleunigen.

Nutzen Sie make  
zur Steuerung  
Ihrer Erstel-  
lungsvorgänge.