Aaron Hillegass

Objective-C Der Einstieg

Der Big Nerd Ranch Guide

ADDISON-WESLEY

ALWAYS LEARNING

PEARSON

Objective-C – der Einstieg

Objective-C – der Einstieg


ADDISON-WESLEY

An imprint of Pearson

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

10 9 8 7 6 5 4 3 2 1

14 13 12

ISBN 978-3-8273-3151-9 (Buch) ; 978-3-8632-4544-3 (pdf) ; 978-3-8632-4171-1 (ePub)

© 2012 by Addison-Wesley Verlag,
ein Imprint der Pearson Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Übersetzung: Jürgen Dubau
Lektorat: Boris Karnikowski, bkarnikowski@pearson.de
Fachlektorat: Matthias Fricke, assense.com
Korrektorat: Marita Böhm, München
Covermotiv: Ellie Volckhausen
Covergestaltung: Marco Lindenbeck, mlindenbeck@webwo.de
Herstellung: Philipp Burkart, pburkart@pearson.de
Satz: mediaService, Siegen (www.mediaService.tv)
Druck und Verarbeitung: Drukarnia Dimograf

Printed in Republic of Poland

Teil II

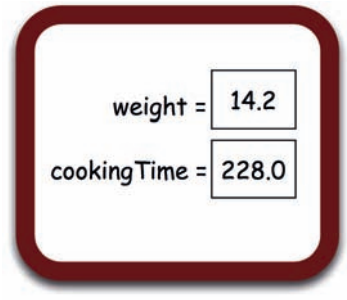
So funktioniert Programmierung

In den nächsten Kapiteln werden Sie viele Programme erstellen, die nützliche Konzepte demonstrieren. Diese Befehlszeilenprogramme sind nichts, mit dem Sie vor Ihren Freunden angeben können, aber es sollte Ihnen einen kleinen Kick geben, wenn Sie sie gemeistert haben, sodass sie starten: Sie vollziehen den Schritt vom Computernutzer zum Computerprogrammierer.

Ihre Programme in diesen ersten Kapiteln werden Sie in C schreiben. Beachten Sie, dass diese Kapitel nicht dazu gedacht sind, diese Sprache detailliert auszuführen. Ganz im Gegenteil: Aus jahrelanger Unterrichtsarbeit hat sich diese wesentliche Auswahl dessen ergeben, was alle Neulinge über Programmierung im Allgemeinen und über C im Besonderen wissen müssen, bevor sie das Programmieren in Objective-C angehen.

3 Variablen und Typen

Greifen wir noch einmal die Metapher mit dem Rezept aus dem letzten Kapitel auf: Köche arbeiten manchmal mit einer kleinen Tafel in der Küche, um Daten zu speichern. Wenn ein Koch z. B. einen Truthahn auspackt, sieht er ein Etikett, auf dem „14.2 pounds“ steht. Bevor er die Verpackung wegwirft, notiert er sich auf der Tafel „Gewicht = 14,2“. Dann berechnet er die Garzeit, kurz bevor er das Geflügel in den Ofen steckt (15 Minuten + 15 Minuten pro Pfund), indem er das auf der Tafel notierte Gewicht hinzuzieht.



▶ *Abbildung 3.1: Mit einer Tafel die Übersicht bei Daten behalten*

Während der Ausführung muss ein Programm oft Daten speichern, die später genutzt werden sollen. Eine Stelle, wo eine bestimmte Art Daten abgelegt wird, nennt man *Variable*. Jede Variable hat einen Namen (wie `cookingTime`) und einen Typ (z. B. eine Zahl). Außerdem wird die Variable bei Ausführung des Programms einen Wert bekommen (z. B. 228.0). Bitte beachten Sie, dass nicht wie im Deutschen das Komma das Dezimalzeichen ist, sondern der Punkt!

3.1 Typen

In einem Programm erstellen Sie eine neue Variable, indem Sie deren Typ und Namen *deklarieren*. Hier folgt das Beispiel für die Deklaration einer Variablen:

```
float weight;
```

Der Typ dieser Variablen ist `float` und ihr Name `weight`. An diesem Punkt hat die Variable keinen Wert.

3.1 Typen

In C müssen Sie aus zwei Gründen den Typ jeder Variablen deklarieren:

- Durch Angabe des Typs kann der Compiler Ihre Arbeit prüfen und weist Sie auf mögliche Fehler oder Probleme hin. Nehmen wir beispielsweise an, dass Sie eine Variable mit einem Typ haben, der Text enthält. Wenn Sie nach deren Logarithmus fragen, weist Sie der Compiler etwa darauf hin: „Es ist sinnlos, den Logarithmus dieser Variablen abzufragen."
- Aus dem Typ kann der Compiler entnehmen, wie viel Platz im Arbeitsspeicher (wie viele Bytes) er für diese Variable reservieren muss.

Hier folgt eine Übersicht über häufig verwendete Typen. In späteren Kapiteln werden wir uns eingehender mit jedem Typ beschäftigen.

- `short`, `int`, `long`

  Diese drei Typen sind ganze Zahlen und benötigen kein Dezimalkomma. `short` benötigt normalerweise weniger Bytes Speicherplatz als `long`, und `int` liegt dazwischen. Somit können Sie in `long` eine weitaus größere Zahl speichern als in `short`.

- `float`, `double`

  Ein `float` ist eine Gleitkommazahl, also eine Zahl, die ein Dezimalkomma haben kann. Im Speicher wird `float` als Mantisse und als Exponent gespeichert. Die Zahl 346,2 wird beispielsweise als $3{,}462 \times 10^2$ repräsentiert. Ein `double` ist eine 64-Bit-Gleitkommazahl (*double precision*), die üblicherweise mehr Bits aufweist, um eine längere Mantisse und größere Exponenten aufzunehmen.

- `char`

  Ein `char` ist eine 1-Byte-Integerzahl, die normalerweise als Zeichen behandelt wird, z. B. der Buchstabe `'a'`.

- Zeiger

  Zeiger enthalten eine Speicheradresse. Sie werden mit dem Sternchen (*Asterisk*) deklariert. Eine als `int *` deklarierte Variable kann z. B. eine Speicheradresse enthalten, in der ein `int` gespeichert wird. Darin ist nicht der eigentliche Wert der Zahl enthalten, aber wenn Sie die Adresse des `int` kennen, können Sie ganz einfach an seinen Wert kommen. Zeiger sind sehr hilfreich, und später werden wir uns eingehender damit beschäftigen. Sehr eingehend!

- `struct`

  Ein `struct` (steht für Struktur) ist ein Typ, der sich aus anderen Typen zusammensetzt. Sie können auch neue `struct`-Definitionen erstellen. Stellen Sie sich z. B. vor, Sie brauchen einen Typ namens `GeoLocation`, der die beiden Gleitkommazahlen für `latitude` und `longitude` enthält. Für diesen Fall würden Sie einen `struct`-Typ definieren.

Mit diesen Typen arbeitet ein C-Programmierer täglich. Es ist recht erstaunlich, welche komplexen Ideen man anhand dieser fünf einfachen Konzepte abbilden kann.

3.2 Ein Programm mit Variablen

Wieder zurück in **Xcode** werden Sie ein weiteres Projekt erstellen. Schließen Sie zuerst das Projekt **AGoodStart**, damit Sie nicht aus Versehen neuen Code in das alte Projekt eintippen.

Nun erstellen Sie ein neues Projekt (**File → New → New Project**). Dieses Projekt wird ein **C Command Line Tool** namens **Turkey**.

Im Projektnavigator wechseln Sie zur Datei `main.c` dieses Projekts und öffnen sie. Bearbeiten Sie `main.c`, sodass der folgende Code darin steht:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    // Deklariert Variable als Typ float und nennt sie 'weight'
    float weight;

    // Zahl in Variable eingeben
    weight = 14.2;

    // Für Benutzer loggen
    printf("The turkey weighs %f.\n", weight);

    // Weitere Variable als Typ float deklarieren
    float cookingTime;

    // Garzeit berechnen und in Variable speichern
    // In diesem Fall steht '*' für 'multipliziert mit'
    cookingTime = 15.0 + 15.0 * weight;

    // Für Benutzer loggen
    printf("Cook it for %f minutes.\n", cookingTime);

    // Funktion beenden und Erfolgsmeldung ausgeben

    return 0;
}
```

Erstellen Sie das Programm und starten Sie es. Sie können entweder oben links im **Xcode**-Fenster auf **Run** klicken oder das Tastenkürzel cmd-R nehmen. Klicken Sie dann auf die Schaltfläche, um zum Protokollnavigator zu gelangen. Wählen Sie das Item oben mit der Bezeichnung **Debug Turkey**, um den Output zu sehen. Der sollte wie folgt aussehen:

```
The turkey weighs 14.200000.
Cook it for 228.000000 minutes.
```

Nun klicken Sie auf die Schaltfläche , um zum Projektnavigator zurückzukehren. Dann wählen Sie `main.c`, damit Sie den Code wieder sehen können. Gehen wir noch einmal durch, was Sie gemacht haben.

In der Programmzeile mit

```
float weight;
```

sagen wir, dass „die Variable `weight` als Typ `float` deklariert wird".

In der nächsten Zeile bekommt diese Variable einen Wert:

```
weight = 14.2;
```

Sie kopieren Daten in diese Variable. Wir sagen, dass „dieser Variablen der Wert 14.2 zugewiesen wird".

In modernem C können Sie in einer Zeile eine Variable deklarieren und gleich einen Anfangswert zuweisen:

```
float weight = 14.2;
```

Hier ist eine weitere Zuweisung:

```
cookingTime = 15.0 + 15.0 * weight;
```

Alles, was rechts vom = steht, ist ein *Ausdruck* (*expression*). Ein Ausdruck ist etwas, was ausgewertet wird und zu einem bestimmten Wert führt. Tatsächlich hat jede Zuweisung rechts vom = einen Ausdruck.

In dieser Zeile

```
weight = 14.2;
```

lautet der Ausdruck z. B. einfach `14.2`.

Variablen sind die Bausteine aller Programme. Dies ist nur eine Einführung in die Welt der Variablen. Sie werden im weiteren Verlauf des Buches mehr darüber erfahren, wie Variablen funktionieren und wie man mit ihnen arbeitet.

3.3 Aufgabe

Erstellen Sie ein neues **C Command Line Tool** namens **TwoFloats**. In dessen Funktion `main()` deklarieren Sie zwei Variablen des Typs `float` und weisen jeder eine Dezimalzahl mit Komma zu, z. B. 3,14 oder 42,0. Deklarieren Sie eine andere Variable des Typs `double` und weisen Sie ihr die Summe der beiden `float`s zu. Geben Sie das Ergebnis mit `printf()` aus. Schauen Sie sich ggf. den Code in diesem Kapitel noch einmal an, wenn Sie die Syntax prüfen müssen.

4 if/else

Ein wichtiges Konzept beim Programmieren ist, situationsabhängig verschiedene Aktionen auszuführen. Sind alle Felder im Rechnungsformular ausgefüllt? Falls ja, soll die Schaltfläche **Senden** aktiv werden. Hat der Spieler noch Leben übrig? Ist das der Fall, soll das Spiel fortgesetzt werden. Falls nicht, soll das Bild des Sargs gezeigt und die traurige Musik abgespielt werden.

Diese Art Verhalten wird anhand von `if` and `else` implementiert. Deren Syntax ist wie folgt:

```
if (Bedingung) {
    // Diesen Code ausführen, wenn die Bedingung wahr ist
} else {
   // Diesen Code ausführen, wenn die Bedingung falsch ist
}
```

In diesem Kapitel werden Sie kein Projekt erstellen. Stattdessen sollten Sie die Codebeispiele sorgfältig durchdenken und das berücksichtigen, was Sie in den letzten beiden Kapiteln gelernt haben.

Hier folgt ein Codebeispiel für `if` und `else`:

```
float truckWeight = 34563.8;

// Liegt das Gewicht unter dem Limit?
if (truckWeight < 40000.0) {
    printf("It is a light truck\n");
} else {
    printf("It is a heavy truck\n");
}
```

Wenn es keine `else`-Klausel gibt, können Sie den Teil weglassen:

```
float truckWeight = 34563.8;

// Liegt das Gewicht unter dem Limit?
if (truckWeight < 40000.0) {
    printf("It is a light truck\n");
}
```

Der Bedingungsausdruck ist entweder wahr oder falsch. In C hat man festgelegt, dass 0 für falsch steht, und alles, was nicht null ist, wird dann als wahr betrachtet.

Im obigen Beispiel akzeptiert der <-Operator eine Zahl auf jeder Seite. Wenn die Zahl auf der linken Seite kleiner ist als die auf der rechten, wird der Ausdruck zu 1 ausgewertet (ein sehr üblicher Weg, um etwas als wahr auszudrücken). Wenn die Zahl auf der linken Seite größer oder gleich der Zahl rechts ist, dann ergibt der Ausdruck 0 (der einzige Weg, um etwas als falsch auszudrücken).

Operatoren erscheinen oft in bedingten Ausdrücken. *Tabelle 4.1: Vergleichsoperatoren* zeigt häufig verwendete Operatoren für den Vergleich von Zahlen (und anderen Typen, die der Computer als Zahlen auswertet):

<	Ist die Zahl auf der linken Seite kleiner als die auf der rechten?
>	Ist die Zahl auf der linken Seite größer als die auf der rechten?
<=	Ist die Zahl auf der linken Seite kleiner oder gleich der auf der rechten?
>=	Ist die Zahl auf der linken Seite größer oder gleich der auf der rechten?
==	Sind beide gleich?
!=	Sind beide nicht gleich?

Tabelle 4.1: Vergleichsoperatoren

Der Operator == verdient noch eine weitere Anmerkung: Beim Programmieren wird mit diesem Operator auf Gleichheit geprüft. Mit einem einzelnen = wird *ein Wert zugewiesen*. Sehr viele Bugs stammen daher, dass Programmierer dort ein = geschrieben haben, wo eigentlich ein == gemeint war. Somit sollten Sie am besten damit aufhören, = als „Gleichheitszeichen" zu betrachten. Ab jetzt sollten Sie davon als „Zuweisungsoperator„ sprechen.

Einige bedingte Ausdrücke erfordern logische Operatoren. Wie wäre es z. B., wenn Sie wissen wollen, ob sich eine Zahl in einem bestimmten Bereich befindet, also etwa größer als null ist oder kleiner als 40.000? Um einen Bereich festzulegen, nutzen Sie den logischen AND-Operator (&&):

```
if ((truckWeight > 0.0) && (truckWeight < 40000.0)) {
    printf("Truck weight is within legal range.\n");
}
```

Tabelle 4.2: Logische Operatoren zeigt die drei logischen Operatoren:

&&	Logisches AND – wahr, wenn und nur wenn beides wahr ist
||	Logisches OR – falsch, wenn und nur wenn beides falsch ist
!	Logisches NOT – wahr wird falsch, falsch wird wahr

Tabelle 4.2: Logische Operatoren

(Falls Sie in einer anderen Sprache bewandert sind, sollten Sie hier beachten, dass es bei Objective-C kein logisches exklusives OR gibt und wir dies somit hier auch nicht erläutern.)

Der logische NOT-Operator (!) negiert den Ausdruck, der rechts von ihm in Klammern steht.

```
// Ist das Gewicht nicht im erlaubten Bereich?
if (!((truckWeight > 0.0) && (truckWeight < 40000.0))) {
    printf("Truck weight is not within legal range.\n");
}
```

4.1 Boolesche Variablen

Wie Sie sehen, können Ausdrücke recht lang und komplex werden. Manchmal ist es hilfreich, den Wert des Ausdrucks in eine praktische und gut benannte Variable zu legen.

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight <  40000.0));
if (isNotLegal) {
    printf("Truck weight is not within legal range.\n");
}
```

Eine Variable, die wahr oder falsch sein kann, bezeichnet man als *boolesche* Variable. Historisch gesehen haben C-Programmierer für boolesche Werte stets ein `int` benutzt. Objective-C-Programmierer nehmen üblicherweise dafür den Datentyp `BOOL`, und das machen wir hier auch. (`BOOL` ist einfach ein anderer Name für einen Integertyp.) Um `BOOL` in einer C-Funktion zu verwenden, müssen Sie den passenden Header einfügen:

```
#include <objc/objc.h>
```

Anmerkung zur Syntax: Wenn der auf den bedingten Ausdruck folgende Code nur aus einer Anweisung besteht, sind die geschweiften Klammern optional. Also entspricht der folgende Code dem vorigen Beispiel.

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight <  40000.0));
if (isNotLegal)
    printf("Truck weight is not within legal range.\n");
```

Wenn der Code aus mehr als einer Anweisung besteht, sind die geschweiften Klammern allerdings notwendig.

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight <  40000.0));
if (isNotLegal) {
    printf("Truck weight is not within legal range.\n");
    printf("Impound truck.\n");
}
```

Warum? Stellen Sie sich mal vor, Sie entfernen die geschweiften Klammern.

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight <  40000.0));
if (isNotLegal)
    printf("Truck weight is not within legal range.\n");
    printf("Impound truck.\n");
```

Mit einem solchen Code machen Sie sich bei den Truckern sehr unbeliebt. In diesem Fall wird jeder Truck beschlagnahmt – egal wie viel er wiegt. Wenn der Compiler nach der Bedingung keine geschweifte Klammer findet, wird nur die nächste Anweisung als zum `if`-Konstrukt gehörig betrachtet. Somit wird die zweite Anweisung immer ausgeführt. (Warum ist die zweite Anweisung eingerückt? Für die menschlichen Leser des Codes ist die Einrückung sehr hilfreich, dem Compiler ist sie egal.)

4.2 else if

Was machen Sie, wenn es mehr als zwei Möglichkeiten gibt? Sie können sie anhand von `else if` der Reihe nach prüfen. Stellen wir uns einmal vor, dass ein Truck zu einer von drei Gewichtskategorien gehört: schwebend, leicht und schwer.

```
if (truckWeight <= 0) {
    printf("A floating truck\n");
} else if (truckWeight < 40000.0) {
    printf("A light truck\n");
} else {
    printf("A heavy truck\n");
}
```

Sie können beliebig viele `else if`-Klauseln nutzen. Alle werden in der Reihenfolge geprüft, in der sie erscheinen, bis eine dann als wahr ausgewertet wird. Die Formulierung „in der Reihenfolge, in der sie erscheinen“ ist wichtig. Achten Sie darauf, dass Sie die Bedingungen richtig ordnen, damit Sie keine falschen Positive bekommen. Wenn Sie im obigen Beispiel etwa die beiden ersten Tests austauschen, dann werden Sie nie einen schwebenden Truck finden, weil diese Art Trucks auch leichte Trucks sind. Die finale `else`-Klausel ist optional, aber praktisch, wenn Sie alles abdecken wollen, was die bisher genannten Bedingungen nicht erfüllt.

4.3 Wenn Sie noch mehr wissen wollen: Bedingungsoperatoren (ternäre Operatoren)

Es ist nicht unüblich, dass man den Wert einer Instanzvariablen mit `if` und `else` setzt. Sie könnten z. B. auf folgenden Code treffen:

```
int minutesPerPound;
if (isBoneless)
    minutesPerPound = 15;
else
    minutesPerPound = 20;
```

Sobald Sie es mit einem Szenario zu tun haben, wo einer Variablen ein Wert basierend auf einer Bedingung zugewiesen wird, haben Sie einen Kandidaten für den *Bedingungsoperator*, der `?` lautet. (Manchmal wird er auch *ternärer Operator* genannt.)

```
int minutesPerPound = isBoneless ? 15 : 20;
```

Diese eine Zeile entspricht dem obigen Beispiel. Anstatt `if` und `else` zu schreiben, formulieren Sie es als Zuweisung. Der Teil vor dem `?` ist die Bedingung. Die Werte nach dem `?` sind die Alternativen dafür, wenn bei der Bedingung wahr oder falsch festgestellt wird.

Wenn diese Notation Ihnen komisch vorkommt, können Sie auch ruhig stattdessen bei `if` und `else` bleiben – daran ist nichts verkehrt. Ich gehe stark davon aus, dass Sie im Laufe der Zeit den ternären Operator als kurze und bündige Schreibweise für eine bedingte Wertezuweisung akzeptieren werden. Wichtiger noch – Sie werden es auch bei anderen Programmierern sehen, und dann ist es sehr angenehm zu verstehen, was Sie da sehen!

4.4 Aufgabe

Schauen Sie sich das folgende Code-Snippet an:

```
int i = 20;
int j = 25;

int k = ( i > j ) ? 10 : 5;

if ( 5 < j - k ) { // Erster Ausdruck
    printf("The first expression is true.");
} else if ( j > i ) { // Zweiter Ausdruck
    printf("The second expression is true.");
} else {
    printf("Neither expression is true.");
}
```

Was wird auf der Konsole ausgegeben?

5 Funktionen

In Kapitel 3 habe ich das Konzept der Variablen vorgestellt: ein Name, der mit bestimmten Daten verbunden ist. Eine Funktion ist ein Name, der mit bestimmtem Code verknüpft ist. Sie können Informationen an eine Funktion übergeben. Sie können die Funktion Code ausführen lassen. Sie können bei einer Funktion auch dafür sorgen, dass Informationen an Sie zurückgegeben werden.

Funktionen sind für das Programmieren fundamental, und somit gibt es davon eine Menge in diesem Kapitel: drei neue Projekte, ein neues Tool und viele neue Konzepte und Ideen. Fangen wir mit einer Übung an, die demonstriert, wofür Funktionen gut sind.

5.1 Wann sollte ich mit einer Funktion arbeiten?

Nehmen wir an, Sie schreiben ein Programm, um Teilnehmern dafür zu danken, dass sie an einem Seminar auf der Big Nerd Ranch teilgenommen haben. Bevor Sie sich damit abmühen, die Teilnehmerliste aus einer Datenbank zu beschaffen oder gute Briefbogen der Big Nerd Ranch mit dem Ausdruck der Zertifikate zu verbraten, sollten Sie mit der Nachricht experimentieren, die auf die Zertifikate gedruckt werden soll.

Für dieses Experiment erstellen Sie ein neues Projekt: ein **C Command Line Tool** namens **ClassCertificates**.

Ihre ersten Überlegungen, ein solches Programm zu schreiben, könnten wie folgt aussehen:

```
int main (int argc, const char * argv[])
{
    printf("Mark has done as much Cocoa Programming as I could fit into 5 days\n");
    printf("Bo has done as much Objective-C Programming as I could fit into 2 days\n");
    printf("Mike has done as much Python Programming as I could fit into 5 days\n");
    printf("Ted has done as much iOS Programming as I could fit into 5 days\n");

    return 0;
}
```

Kriegen Sie schon Kopfschmerzen, wenn Sie nur daran denken, all das einzutippen? Sind all diese Wiederholungen nicht nervig? Wenn Sie dies bejahen, haben Sie das Zeug zu einem hervorragenden Pro-

grammierer. Wenn Sie auf immer wiederkehrende Arbeit treffen, die sich von der Art her sehr ähnelt (in diesem Fall die Wörter in der `printf`-Anweisung), sollten Sie sich überlegen, dass man die gleiche Aufgabe besser mit einer Funktion erledigen kann.

5.2 Wie schreibe und nutze ich Funktionen?

Da Sie nun erkannt haben, dass Sie eine Funktion brauchen, sollten Sie auch eine schreiben. Öffnen Sie `main.c` im Projekt **ClassCertificates** und fügen Sie vor der Funktion `main` eine neue Funktion ein. Geben Sie ihr den Namen `congratulateStudent`.

```
#include <stdio.h>

void congratulateStudent(char *student, char *course, int numDays)
{
    printf("%s has done as much %s Programming as I could fit into %d days.\n",
         student, course, numDays);
}
```

(Neugierig, was `%s` und `%d` bedeuten? Bitte noch etwas Geduld, das ist Thema des nächsten Kapitels.)

Nun bearbeiten Sie `main`, um diese neue Funktion nutzen zu können:

```
int main (int argc, const char * argv[])
{
   congratulateStudent("Mark", "Cocoa", 5);
   congratulateStudent("Boris", "Objective-C", 2);
   congratulateStudent("Mike", "Python", 5);
   congratulateStudent("Ted", "iOS", 5);

    return 0;
}
```

Kompilieren Sie das Programm und starten Sie es. Sie bekommen wahrscheinlich einen *Warnhinweis*: ein Ausrufezeichen in einem kleinen gelben Dreieck. Eine Warnung in **Xcode** verhindert nicht, dass Ihr Programm läuft, sondern soll nur Ihre Aufmerksamkeit auf ein mögliches Problem richten. Der Text der Warnung steht rechts neben dem Code. Diese Warnung lautet etwa wie folgt: `No previous prototype for function 'congratulateStudent'` (Bisher kein Prototyp für Funktion 'congratulateStudent'). Ignorieren Sie für den Augenblick diese Warnung, da wir uns am Ende dieses Abschnitts darum kümmern werden.

Suchen Sie den Output im Protokollnavigator. Da sollte genau das stehen, was Sie auch angezeigt bekämen, wenn Sie alles selbst eingetippt hätten.

```
Mark has done as much Cocoa Programming as I could fit into 5 days.
Bo has done as much Objective-C Programming as I could fit into 2 days.
Mike has done as much Python Programming as I could fit into 5 days.
Ted has done as much iOS Programming as I could fit into 5 days.
```

Gehen Sie gedanklich noch einmal durch, was Sie hier gemacht haben. Ihnen ist ein sich wiederholendes Muster aufgefallen. Sie haben die gemeinsamen Kennzeichen des Problems aufgegriffen (den sich wiederholenden Text) und sie in eine separate Funktion verschoben. So blieben nur noch die Unterschiede übrig (Teilnehmername, Seminarname, Anzahl Tage). Sie haben sich um diese Unterschiede gekümmert, indem Sie der Funktion drei *Parameter* hinzugefügt haben. Schauen wir uns noch einmal die Zeile an, in der Sie die Funktion benennen.

```
void congratulateStudent(char *student, char *course, int numDays)
```

Jeder Parameter besteht aus zwei Teilen: aus dem Datentyp, den das Argument repräsentiert, und dem Namen des Parameters. Parameter werden durch Komma getrennt und in Klammern rechts neben den Namen der Funktion gestellt.

Was soll das `void` links neben dem Funktionsnamen? Das ist die Art Information, die durch die Funktion zurückgegeben wird. Wenn Sie keine Information haben, die zurückgegeben werden soll, nehmen Sie das Schlüsselwort `void`. Weiter hinten in diesem Kapitel werden wir uns eingehender mit dem Zurückgeben beschäftigen.

Sie haben außerdem die neue Funktion in `main` verwendet bzw. *aufgerufen*. Als `congratulateStudent` aufgerufen wurde, haben Sie dieser Funktion Werte übergeben. Werte, die einer Funktion übergeben werden, nennt man *Argumente*. Der Wert dieses Arguments wird dann dem entsprechenden Parameternamen zugewiesen. Diesen Parameternamen kann man innerhalb der Funktion als Variable nutzen, die den übergebenen Wert enthält.

Schauen wir uns das nun genauer an. Im ersten Aufruf von `congratulateStudent` übergeben Sie drei Argumente: `"Mark"`, `"Cocoa"`, `5`.

```
    congratulateStudent("Mark", "Cocoa", 5);
```

Hier wollen wir uns nun auf das dritte Argument konzentrieren. Wenn `5` an `congratulateStudent` übergeben wird, wird es dem dritten Parameter `numDays` zugewiesen. Argumente und Parameter werden einander in der Reihenfolge zugeordnet, wie sie erscheinen. Sie müssen also vom gleichen Datentyp (oder sehr nahe verwandt) sein. Hier ist `5` ein Integerwert, und der Typ von `numDays` ist `int`. Gut.

Wenn `congratulateStudent` nun die Variable `numDays` innerhalb der Funktion verwendet, also *referenziert*, wird dessen Wert `5` betragen. Sie können sehen, dass `numDays` direkt vor dem Semikolon referenziert wird. Schließlich können Sie sich überzeugen, dass alles funktioniert hat, indem Sie sich die erste Zeile des Outputs anschauen, in der die Zahl der Tage korrekt dargestellt wird.

Schauen Sie sich noch einmal unseren ersten Vorschlag der Version für **ClassCertificates** mit all dem wiederholten Eintippen an. Was bringt es denn nun, stattdessen eine Funktion zu nehmen? Sich Tipparbeit zu

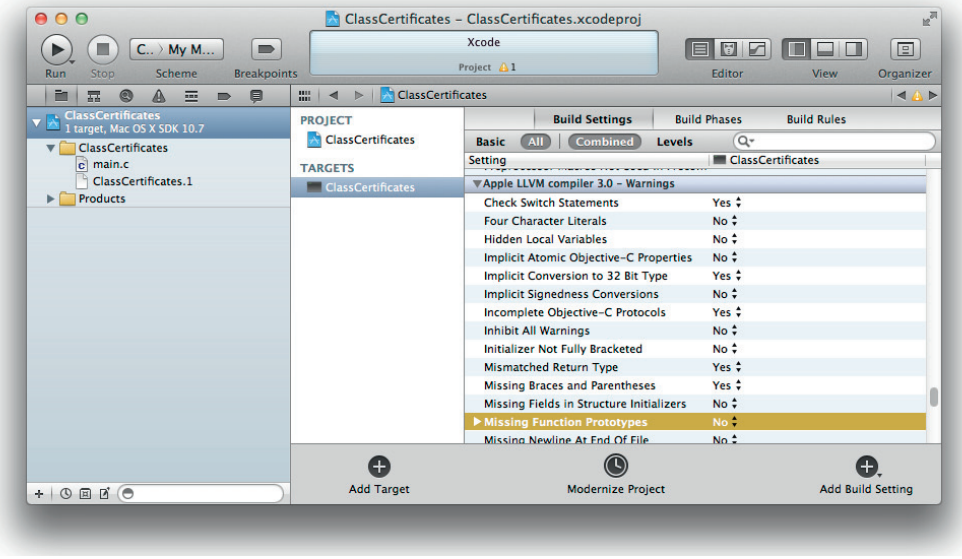
ersparen? Nun, das stimmt, ist aber definitiv noch nicht alles. Es geht auch um Fehlerprüfung. Je weniger Sie tippen und je mehr der Computer die Verarbeitung der Zahlen übernehmen muss, desto weniger Tippfehler passieren. Wenn Sie sich außerdem bei einem Funktionsnamen vertippen, werden Sie von **Xcode** darauf hingewiesen, aber **Xcode** hat keine Ahnung, wenn Sie sich bei normalem Text vertippen.

Ein weiterer Vorteil der Funktionen ist die Wiederverwendbarkeit. Nachdem Sie diese praktische Funktion geschrieben haben, können Sie sie in einem anderen Programm erneut einsetzen. Änderungen werden so ebenfalls einfacher. Sie brauchen einfach nur die Dankesformulierung an einer Stelle anpassen, und dann wirkt sie sich überall aus.

Der finale Vorteil von Funktionen ist: Falls es einen „Bug" gibt, können Sie diese eine Funktion reparieren, und plötzlich wird alles, was davon aufgerufen wird, korrekt funktionieren. Wenn Sie Ihren Code in Funktionen aufteilen, wird er einfacher zu verstehen und zu pflegen.

Nun wenden wir uns der Warnung im Code zu. Es ist sehr üblich, eine Funktion an einer Stelle zu *deklarieren* und an einer anderen zu *definieren*. Die Deklaration einer Funktion warnt den Compiler nur vor, dass gleich eine Funktion mit einem bestimmten Namen kommt. Die auszuführenden Schritte beschreiben Sie in der Funktion, die definiert wird. In dieser Übung haben Sie tatsächlich die Funktion an der gleichen Stelle deklariert und definiert. Weil das unüblich ist, gibt **Xcode** eine Warnung aus, falls Ihre Funktion nicht vorab deklariert wurde.

Hier ist es okay, diese Warnung bei allen Projekten zu ignorieren, die Sie in diesem Buch erstellen. Oder Sie nehmen sich die Zeit, sie zu deaktivieren. Dafür wählen Sie das Ziel **ClassCertificates**, das Sie ganz oben im Projektnavigator finden. Im Editorbereich wählen Sie **All** auf dem Reiter **Build Settings**. Scrollen Sie durch die verschiedenen Build-Einstellungen und suchen Sie **Missing Function Prototypes**. Ändern Sie diese Einstellung auf **No**.

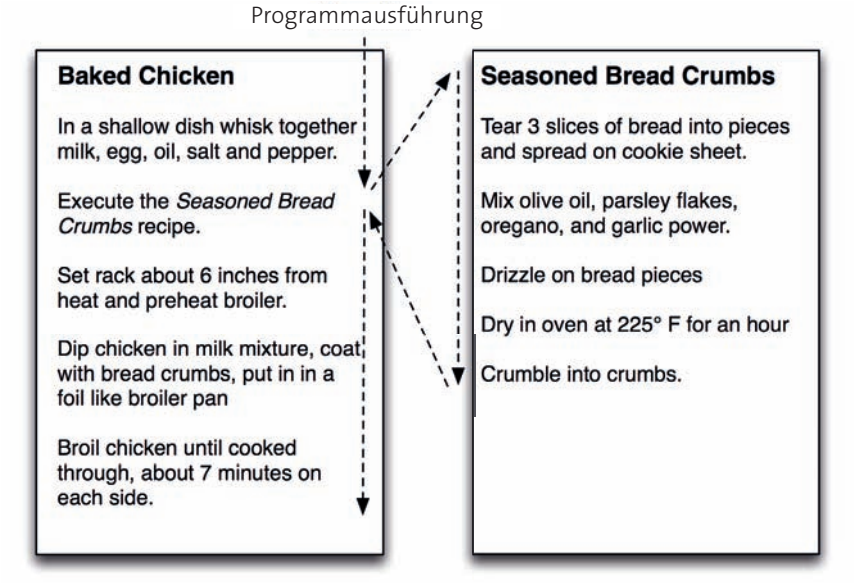


▶ *Abbildung 5.1: Deaktivieren der Warnung* **Missing Function Prototypes**

5.3 Wie Funktionen zusammenarbeiten

Ein *Programm* ist eine Sammlung von Funktionen. Wenn man ein Programm startet, werden diese Funktionen von der Festplatte in den Arbeitsspeicher kopiert. Dann sucht der Prozessor die Funktion „main" und führt sie aus.

Erinnern Sie sich daran, dass eine Funktion wie eine Rezeptanweisung ist. Wenn ich die Karte „Baked Chicken" ausführen will, entdecke ich, dass die zweite Anweisung lautet: „Make Seasoned Bread Crumbs", was auf einer anderen Karte erläutert wird. Ein Programmierer würde es so ausdrücken: „Die Funktion *Baked Chicken* ruft die Funktion *Seasoned Bread Crumbs* auf."



▶ *Abbildung 5.2: Rezeptkarten*

Entsprechend kann die Funktion `main` andere Funktionen aufrufen. Zum Beispiel hat Ihre `main`-Funktion in **ClassCertificates** die Funktion `congratulateStudent` aufgerufen, die wiederum dann `printf` aufruft.

Wenn Sie die gewürzten Brotkrumen vorbereiten, unterbrechen Sie die Ausführung der Karte „Baked Chicken". Wenn die Brotkrumen fertig sind, setzen Sie die Ausführung der Karte „Baked Chicken" wieder fort.

Entsprechend unterbricht die Funktion `main` die Ausführung und „blockiert", bis die von ihr aufgerufene Funktion mit ihrer Ausführung fertig ist. Um das sehen zu können, werden wir eine Funktion namens `sleep` aufrufen, die nichts anderes macht, als ein paar Sekunden zu warten. In der Funktion `main` fügen Sie den Aufruf von `sleep` ein.

```
int main (int argc, const char * argv[])
{
    congratulateStudent("Mark", "Cocoa", 5);
    sleep(2);
    congratulateStudent("Boris", "Objective-C", 2);
    sleep(2);
    congratulateStudent("Mike", "Python", 5);
    sleep(2);
    congratulateStudent("Ted", "iOS", 5);

    return 0;
}
```

Kompilieren Sie das Programm und starten Sie es. (Ignorieren Sie hier die Warnung über eine implizite Deklaration.) Sie sollten eine zweisekündliche Pause zwischen jeder Gratulation feststellen. Das liegt daran, dass die `main`-Funktion die weitere Ausführung unterbricht, bis die Funktion `sleep` mit der Ruhephase fertig ist.

Beachten Sie, dass Sie zum Aufrufen einer Funktion deren Namen verwenden und die dazugehörigen Argumente in Klammern schreiben. Wenn wir somit also über Funktionen sprechen, werden diese normalerweise mit leeren Klammern geschrieben. Von jetzt an werden wir also `main()` schreiben, wenn von der `main`-Funktion die Rede ist.

Ihr Computer wurde mit vielen eingebauten Funktionen geliefert. Tatsächlich ist das ein wenig irreführend – dies ist die Wahrheit: Bevor Mac OS X auf Ihrem Computer installiert wurde, war er nichts anderes als ein teures Warmluftgebläse. Zu den Dingen, die als Bestandteil von Mac OS X installiert wurden, gehören Dateien, die eine Sammlung von vorkompilierten Funktionen enthalten. Diese Sammlungen nennt man die *Standardbibliotheken*. `sleep()` und `printf()` sind in diesen Standardbibliotheken enthalten.

Ganz oben in `main.c` haben Sie die Datei `stdio.h` eingebunden. Diese Datei enthält eine Deklaration der Funktion `printf()` und lässt den Compiler prüfen, ob Sie sie korrekt verwenden. Die Funktion `sleep()` wird in `stdlib.h` deklariert. Binden Sie diese Datei ebenfalls ein, damit der Compiler sich nicht mehr beschwert, dass `sleep()` implizit deklariert ist:

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

void congratulateStudent(char *student, char *course, int numDays)
{
    ...
```

Die Standardbibliothek dient zweierlei Aufgaben:

- Sie repräsentiert größere Codeblöcke, die Sie nicht erst selbst schreiben und pflegen müssen. Das befähigt Sie, größere und bessere Programme zu erstellen, als wenn Sie alles von Grund auf neu schreiben müssten.
- Außerdem gewährleisten sie, dass die meisten Programme vom Erscheinungsbild und der Handhabung ähnlich sind.

Programmierer verbringen eine Menge Zeit, die Standardbibliotheken jener Betriebssysteme zu studieren, an denen sie arbeiten. Jede Firma, die ein Betriebssystem erstellt, liefert dazu gleich auch eine Dokumentation für die dazugehörigen Standardbibliotheken aus. Sie erfahren in Kapitel 16, wie Sie die Dokumentation für iOS und Mac OS X durchsuchen.

5.4 Lokale Variablen, Frames und der Stack

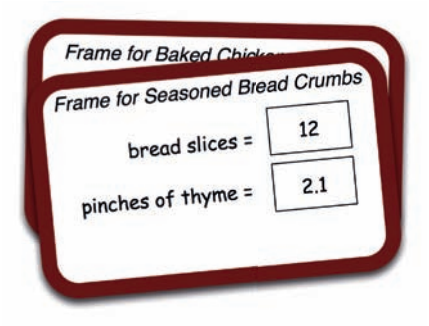
Jede Funktion kann *lokale Variablen* enthalten. Dabei handelt es sich um Variablen, die in einer Funktion deklariert sind. Sie existieren nur während der Ausführung dieser Funktion, und man kann nur in dieser Funktion darauf zugreifen. Nehmen wir beispielsweise an, dass Sie eine Funktion schreiben, die berechnet, wie lange ein Truthahn gegart werden soll. Das könnte wie folgt aussehen:

```
void showCookTimeForTurkey(int pounds)
{
   int necessaryMinutes = 15 + 15 * pounds;
   printf("Cook for %d minutes.\n", necessaryMinutes);
}
```

`necessaryMinutes` ist eine lokale Variable. Sie entstand, als `showCookTimeForTurkey()` mit der Ausführung begann, und hört auf zu existieren, wenn diese Funktion zu Ende geführt wurde. Bei `pounds`, dem Parameter der Funktion, handelt es sich ebenfalls um eine lokale Variable. Ein Parameter ist eine lokale Variable, die mit dem Wert des entsprechenden Arguments initialisiert wurde.

Eine Funktion kann viele lokale Variablen enthalten, und alle werden im *Frame* dieser Funktion gespeichert. Stellen Sie sich den Frame wie eine Tafel vor, auf der Sie schnell etwas notieren können, während die Funktion läuft. Wenn die Funktion abgeschlossen wurde, wird die Tafel wieder abgewischt.

Stellen Sie sich vor, Sie wollen an dem *Baked Chicken*-Rezept arbeiten. In der Küche bekommt jedes Ihrer Rezepte seine eigene Tafel, und so bereiten Sie auch für dieses Rezept eines vor. Wenn Sie nun also das Rezept *Seasoned Bread Crumbs* aufrufen, brauchen Sie eine neue Tafel. Wo legen Sie diese hin? Direkt auf die Tafel für *Baked Chicken*. Immerhin haben Sie die Ausführung von *Baked Chicken* unterbrochen, um *Seasoned Bread Crumbs* zu machen. Sie brauchen das Frame *Baked Chicken* erst dann wieder, wenn das Rezept *Seasoned Bread Crumbs* vollendet ist und dessen Frame entsorgt wird. Sie verfügen nun über einen Stapel Frames (den sogenannten *Stack*).



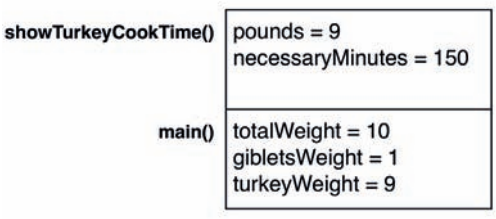
▶ *Abbildung 5.3: Zwei Tafeln auf einem Stack*

Programmierer drücken das so aus: „Wenn eine Funktion aufgerufen wird, wird ihr Frame oben *auf dem Stack* erstellt. Wenn die Funktion beendet ist, wird ihr Frame vom Stack genommen und zerstört."

Schauen wir uns genauer an, wie der Stack funktioniert, indem wir `showCookTimeForTurkey()` in ein hypothetisches Programm einfügen:

```
void showCookTimeForTurkey(int pounds)
{
    int necessaryMinutes = 15 + 15 * pounds;
    printf("Cook for %d minutes.\n", necessaryMinutes);
}

int main(int argc, const char * argv[])
{
    int totalWeight = 10;
    int gibletsWeight = 1;
    int turkeyWeight = totalWeight - gibletsWeight;
    showCookTimeForTurkey(turkeyWeight);
    return 0;
}
```

Erinnern Sie sich daran, dass `main()` stets zuerst ausgeführt wird. `main()` ruft `showCookTimeForTurkey()` auf, das dann mit der Ausführung beginnt. Wie sieht dann der Stack dieses Programms aus, direkt nachdem `pounds` mit 15 multipliziert wurde?



▶ *Abbildung 5.4: Zwei Frames auf dem Stack*

Beim Stack wird zuerst das weggenommen, was zuletzt eingefügt wurde. Man bezeichnet das auch als *last-in, first-out*. Das heißt, die Funktion `showCookTimeForTurkey()` wird zuerst ihren Frame vom Stack nehmen (man nennt das „poppen"), bevor `main()` seinen Frame vom Stack poppt.

Beachten Sie, dass `pounds`, der einzige Parameter von `showCookTimeForTurkey()`, Teil des Frames ist. Erinnern Sie sich, dass ein Parameter eine lokale Variable ist, die dem Wert des zugehörigen Arguments zugewiesen wird. Für dieses Beispiel wird die Variable `turkeyWeight` mit dem Wert `9` als Argument an `showCookTimeForTurkey()` übergeben. Dann wird dieser Wert dem Parameter `pounds` zugewiesen und in den Frame der Funktion kopiert.

5.5 Rekursion

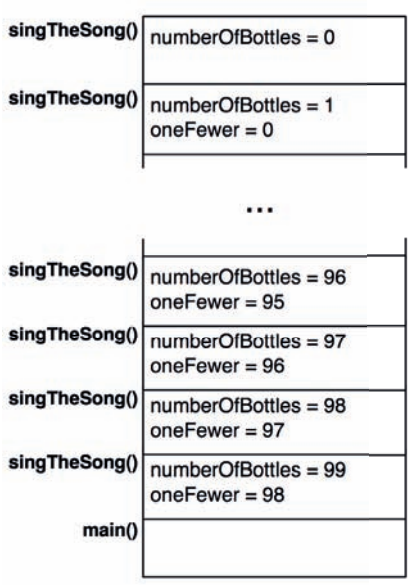
Kann eine Funktion sich selbst aufrufen? Und ob! Das nennt man *Rekursion*. Es gibt einen berüchtigten Song namens „99 Bottles of Beer". Erstellen Sie ein neues **C Command Line Tool** namens **BeerSong**. Öffnen Sie `main.c`, fügen Sie eine Funktion ein, die den Text dieses Liedes ausgibt, und rufen Sie `main()` auf:

```
#include <stdio.h>

void singTheSong(int numberOfBottles)
{
    if (numberOfBottles == 0) {
        printf("There are simply no more bottles of beer on the wall.\n");
    } else {
        printf("%d bottles of beer on the wall. %d bottles of beer.\n",
               numberOfBottles, numberOfBottles);
        int oneFewer = numberOfBottles - 1;
        printf("Take one down, pass it around, %d bottles of beer on the wall.\n",
               oneFewer);
        singTheSong(oneFewer); // Diese Funktion ruft sich selbst auf!
        printf("Put a bottle in the recycling, %d empty bottles in the bin.\n",
               numberOfBottles);
    }
}

int main(int argc, const char * argv[])
{
    singTheSong(99);
    return 0;
}
```

Kompilieren Sie das Programm und starten Sie es. Der Output sieht wie folgt aus:

```
99 bottles of beer on the wall. 99 bottles of beer.
Take one down, pass it around, 98 bottles of beer on the wall.
98 bottles of beer on the wall. 98 bottles of beer.
Take one down, pass it around, 97 bottles of beer on the wall.
97 bottles of beer on the wall. 97 bottles of beer.
...
1 bottles of beer on the wall. 1 bottles of beer.
Take one down, pass it around, 0 bottles of beer on the wall.
There are simply no more bottles of beer on the wall.
Put a bottle in the recycling, 1 empty bottles in the bin.
Put a bottle in the recycling, 2 empty bottles in the bin.
...
Put a bottle in the recycling, 98 empty bottles in the bin.
Put a bottle in the recycling, 99 empty bottles in the bin.
```

Wie sieht der Stack aus, wenn die letzte Flasche aus dem Regal genommen wurde?



▶ *Abbildung 5.5: Frames im Stack einer rekursiven Funktion*

Erläuterungen über Frames und den Stack kommen in einem Programmierkurs für Anfänger normalerweise nicht vor, aber ich bin der Ansicht, dass diese Konzepte für beginnende Programmierer außergewöhnlich hilfreich sind. Erstens bekommt man ein konkreteres Verständnis für Antworten auf Fragen wie „Was geschieht mit meinen lokalen Variablen, wenn die Funktion mit der Ausführung fertig ist?“ Zweitens hilft es, den *Debugger* zu verstehen. Der Debugger ist ein Programm, mit dem Sie besser verstehen, was Ihr Programm eigentlich macht, was umgekehrt auch sehr hilfreich ist, „Bugs“ (also Fehler im Code) zu finden und zu beheben. Wenn Sie ein Programm in **Xcode** erstellen und starten, wird der Debugger an das Programm *angehängt*, damit Sie ihn verwenden können.

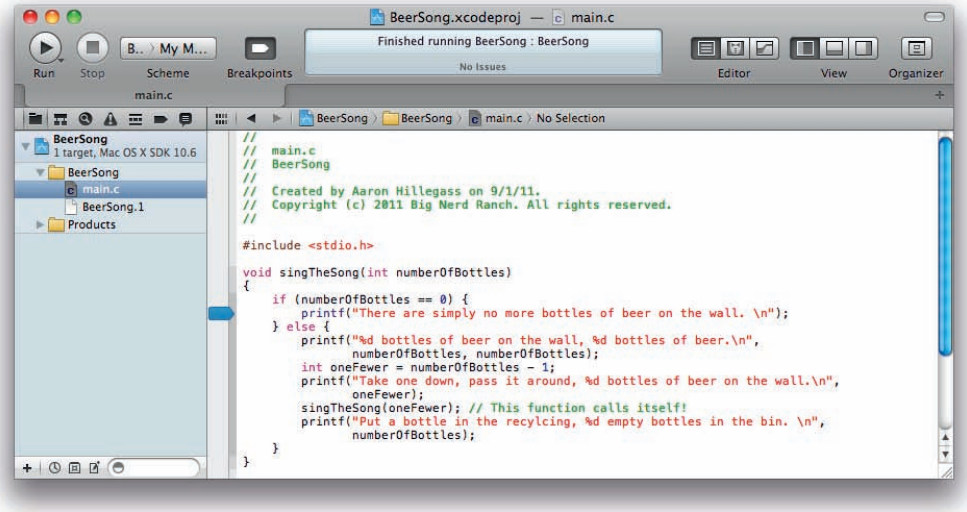
5.6 Die Frames im Debugger untersuchen

Sie können mit dem Debugger die Frames im Stack untersuchen. Dafür müssen Sie allerdings Ihr Programm mitten bei der Ausführung unterbrechen. Anderenfalls wird `main()`bis zum Schluss ausgeführt, und dann hat man keine Frames mehr, die man sich anschauen kann. Um bei unserem Programm **BeerSong** so viele Frames wie möglich zu sehen, soll die Ausführung in der Zeile gestoppt werden, die „There are simply no more bottles of beer on the wall" ausgibt.

Wie wird das gemacht? In `main.c` suchen Sie die Zeile.

```
printf("There are simply no more bottles of beer on the wall.\n");
```

Links neben dem Code befinden sich zwei schattierte Spalten. Klicken Sie auf die breite linke Spalte davon direkt auf Höhe dieser Codezeile.



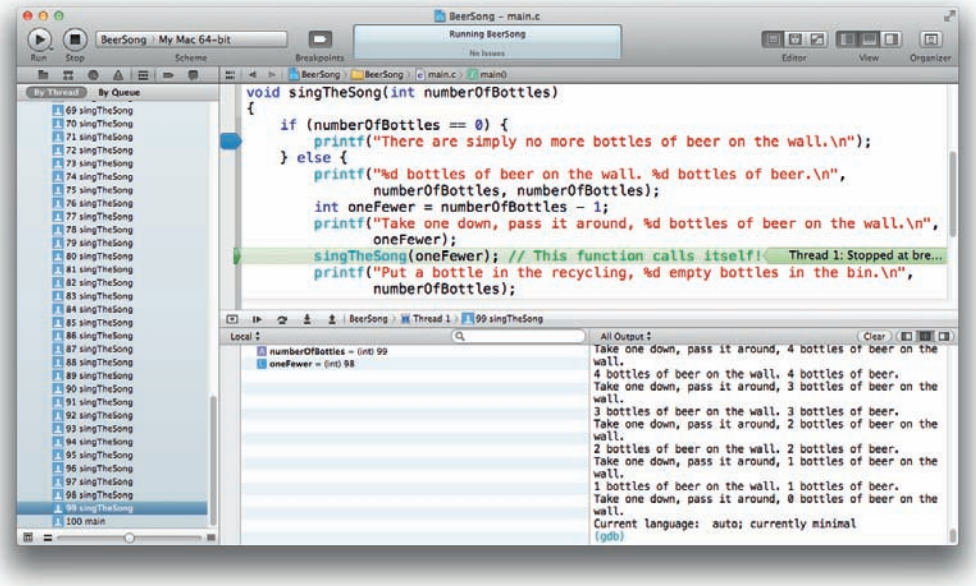
▶ *Abbildung 5.6: Einen Breakpoint setzen*

Der blaue Indikator zeigt, dass Sie einen *Breakpoint* gesetzt haben. Ein Breakpoint ist eine Stelle im Code, wo der Debugger die Ausführung des Programms pausieren soll. Starten Sie das Programm erneut. Es startet und stoppt dann direkt vor Ausführung der Zeile, in der Sie den Breakpoint gesetzt haben.

Damit haben Sie Ihr Programm temporär eingefroren und können es genauer untersuchen. Im Navigationsbereich klicken Sie auf das Icon , um den *Debugnavigator* zu öffnen. Dieser Navigator zeigt alle Frames, die sich aktuell auf dem Stack befinden (auch *Stack Trace* genannt).

Im Stack Trace werden die Frames anhand des Namens ihrer Funktion identifiziert. Angenommen, Ihr Programm besteht fast ausschließlich aus einer rekursiven Funktion, diese Frames haben den gleichen Namen, und Sie müssen sie anhand des Werts von `oneFewer` unterscheiden, die an sie übergeben werden. Unten im Stack finden Sie natürlich den Frame für `main()`.

Sie können einen Frame auswählen, um die Variablen in diesem Frame und den Quellcode für die Codezeile, die aktuell ausgeführt wird, zu sehen. Wählen Sie den Frame für das erste Mal, dass `singTheSong` aufgerufen wird.



► *Abbildung 5.7: Frames im Stack einer rekursiven Funktion*

Sie sehen die Variablen dieses Frames und deren Wert unten links im Fenster. Rechts sehen Sie auch den Output in einem Bereich, der als *Konsole* bezeichnet wird. (Wenn Sie die Konsole nicht sehen, suchen Sie im rechten unteren Bildschirmbereich die Buttons . Klicken Sie auf den mittleren Button, und die Konsole erscheint.) In der Konsole erkennen Sie die Wirkung des Breakpoints: Das Programm wurde unterbrochen, bevor es die Zeile erreicht, die den Song beendet.

Nun müssen wir den Breakpoint entfernen, damit das Programm normal läuft. Ziehen Sie einfach den blauen Indikator aus dem Bildschirm heraus. Oder klicken Sie auf das Icon oben im Navigatorbereich, um den *Breakpoint-Navigator* anzuzeigen, damit Sie alle Breakpoints in einem Projekt sehen. Von dort aus wählen Sie den Breakpoint und löschen ihn.

Um die Ausführung des Programms wieder aufzunehmen, klicken Sie auf den Button in der Debuggerleiste zwischen Editor und Variablenansicht.

Hier haben wir uns nur kurz mit dem Debugger beschäftigt, um zu demonstrieren, wie Frames funktionieren. Doch es ist sehr hilfreich, mit dem Debugger Breakpoints zu setzen und die Frames im Stack eines Programms durchsuchen zu können, wenn sich Ihr Programm nicht wie gewünscht verhält und Sie sich eingehender anschauen wollen, was eigentlich passiert.

5.7 Return

Viele Funktionen geben einen Wert zurück, wenn die Ausführung abgeschlossen ist. Sie entnehmen den Datentyp, den eine Funktion zurückgeben wird, aus dem Typ, der dem Funktionsnamen vorangeht. (Falls eine Funktion nichts zurückgibt, ist der Rückgabetyp `void`.)

Erstellen Sie ein neues **C Command Line Tool** namens **Degrees**. In `main.c` fügen Sie vor `main()` eine Funktion ein, die die Temperatur von Celsius auf Fahrenheit umrechnet. Dann aktualisieren Sie `main()`, um die neue Funktion aufzurufen.

```
#include <stdio.h>

float fahrenheitFromCelsius(float cel)
{
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}

int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
    return 0;
}
```

Sehen Sie, wie wir den Rückgabewert von `fahrenheitFromCelsius()` genommen und ihn der Variablen `freezeInF` vom Typ `float` zugewiesen haben? Ganz schön clever, nicht wahr?

Die Ausführung einer Funktion stoppt, wenn sie zurückgegeben wird. Nehmen wir beispielsweise an, Sie haben folgende Funktion:

```
float average(float a, float b)
{
    return (a + b)/2.0;
    printf("The mean justifies the end\n");
}
```

Wenn Sie diese Funktion aufrufen, wird der Aufruf von `printf()` nie ausgeführt.

Eine naheliegende Frage wäre also: „Warum geben wir aus `main()` immer 0 zurück?“ Wenn Sie 0 ans System zurückgeben, teilen Sie ihm mit: „Alles hat prima funktioniert.“ Wenn Sie das Programm beenden, weil etwas schiefgelaufen ist, geben Sie 1 zurück.

Das scheint dem zu widersprechen, wie 0 und 1 in `if`-Anweisungen funktionieren. Weil 1 für wahr und 0 für falsch steht, wirkt es naheliegend, dass man mit 1 Erfolg und mit 0 Misslingen ausdrückt. Also sollten Sie sich `main()` vorstellen, als werde ein Fehlerbericht zurückgegeben. In diesem Fall ist 0 eine gute Nachricht! Erfolg sind hier ausbleibende Fehler.

Um das klarer zu machen, nutzen einige Programmierer die Konstanten `EXIT_SUCCESS` und `EXIT_FAILURE`, die jeweils 0 bzw. 1 entsprechen. Diese Konstanten werden in der Headerdatei `stdlib.h` definiert:

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

float fahrenheitFromCelsius(float cel)
{
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}

int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
    return EXIT_SUCCESS;
}
```

In diesem Buch werden wir generell 0 anstatt `EXIT_SUCCESS` einsetzen.

5.8 Globale und statische Variablen

In diesem Kapitel haben wir über lokale Variablen gesprochen, die nur während der Ausführung einer Funktion existieren. Es gibt auch Variablen, auf die man jederzeit von einer beliebigen Funktion aus zugreifen kann. Diese werden als *globale Variablen* bezeichnet. Damit eine Variable global wird, deklarieren Sie sie außerhalb einer bestimmten Funktion. Sie könnten z. B. eine Variable `lastTemperature` einfügen, die die Temperatur enthält, die aus Celsius umgerechnet wurde. Fügen Sie dem Programm eine globale Variable hinzu:

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

// Globale Variable deklarieren
float lastTemperature;

float fahrenheitFromCelsius(float cel)
{
    lastTemperature = cel;
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}
int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
    printf("The last temperature converted was %f\n", lastTemperature);
    return EXIT_SUCCESS;
}
```

Jedes komplexe Programm wird Dutzende Dateien mit den verschiedensten Funktionen enthalten. In all diesen Dateien stehen dem Code globale Variablen zur Verfügung. Manchmal wollen Sie, dass eine Variable zwischen verschiedenen Dateien weitergegeben wird. Aber wie Sie sich vorstellen können, kann es auch sehr verwirrend wirken, wenn man es mit einer Variablen zu tun hat, auf die man mit mehreren Funktionen zugreifen kann. Um das zu bewerkstelligen, gibt es *statische Variablen*. Eine statische Variable ist insofern wie eine globale, weil sie außerhalb einer Funktion deklariert wird. Doch auf eine statische Variable kann man nur von dem Code der Datei zugreifen, in der sie deklariert wurde. Also bekommen Sie den nicht standortbezogenen „Existiert außerhalb einer Funktion"-Vorteil und vermeiden gleichzeitig das „Du hast an meiner Variablen herumgefummelt!"-Problem.

Sie können eine globale Variable zu einer statischen machen, aber weil Sie nur eine Datei haben (`main.c`), wird das keine Auswirkungen haben.

```
// Eine statische Variable deklarieren
static float lastTemperature;
```

Sowohl statische als auch globale Variablen können bei der Erstellung einen Anfangswert bekommen:

```
// lastTemperature mit 50 Grad initialisieren
static float lastTemperature = 50.0;
```

Wenn Sie keinen Anfangswert vergeben, werden die Variablen automatisch mit null initialisiert.

In diesem Kapitel haben Sie Funktionen kennengelernt. Wenn wir in Teil III zu Objective-C kommen, werden Sie den Begriff *Methode* hören – eine Methode ist einer Funktion sehr ähnlich.

5.9 Aufgabe

Die Innenwinkel eines Dreiecks müssen zusammen 180 Grad ergeben. Erstellen Sie ein neues **C Command Line Tool** namens **Triangle**. Schreiben Sie in `main.c` eine Funktion, die die ersten beiden Winkel annimmt und den dritten zurückgibt. So wird sie aussehen, wenn Sie sie aufrufen:

```
#include <stdio.h>

// Fügen Sie die neue Funktion hier ein

int main(int argc, const char * argv[])
{
    float angleA = 30.0;
    float angleB = 60.0;
    float angleC = remainingAngle(angleA, angleB);
    printf("The third angle is %.2f\n", angleC);
    return 0;
}
```

Der Output sollte so lauten:

```
The third angle is 90.00
```

Copyright