Eine Einführung in C-Funktionen

CGK-Proseminar 2014

Philip Gawehn

04.07.2014, Hamburg

Inhaltsverzeichnis

1 Einleitung

Funktionen: Sie haben in verschieden Sprachen verschiedene Bezeichnungen. In Java heißen sie Methoden, in C spricht man von Funktionen. Keine Programmiersprache kann ohne sie existieren.

Doch was sind eigentlich Funktionen? Wenn man eine Funktion genauer betrachtet, so kann man erkennen, dass sie nichts anderes ist als ein Unterprogramm, bei dem der sequentielle Ablauf unterbrochen wird. Dabei wird immer dasselbe Grundprinzip verfolgt[1]:

1. Das Programm springt zum ausführbaren Code der Funktion, danach wird
2. der Code der Funktion ausgeführt, bevor
3. das Programm zur Aufrufstelle zurückspringt und den nachfolgenden Code ausführt.

Funktionen werden auch dazu genutzt, um Code, der oft wiederverwendet werden kann (und ein bestimmtes Ziel verfolgt), an einer zentralen Stelle zu lagern. Dabei ist er oft übersichtlicher und, wenn Fehler vorliegen, in der Regel auch einfacher zu korrigieren.
Funktionen können auch in anderen Programmen oder Modulen verwendet werden. Dies findet oft Verwendung in der modularen Programmierung. Ein Beispiel hierfür wäre die String-Library.

2 Grundlagen

2.1 Der Aufbau einer Funktion

```
1 schluesselwoerter rueckgabetyp funktionsname (parameterliste)
2 {
3     Anweisungen
4 }
```

1. Am Anfang der Deklaration (1. Zeile), welche zum Anmelden der Funktion beim Compiler dient, befindet sich eine sinnvolle Kombination der Schlüsselwörter *static*, *extern* und *inline*.
2. Es folgt nun der Typ, den die Funktion zurückgeben soll. *void* wird angegeben, wenn die Funktion nichts zurückgibt.
3. Der Funktionsname ist case-sensitive (d.h.: die Groß-/Kleinschreibung wird beachtet) und darf nur aus den Zeichen '0-9', 'a-Z' und/oder '_' bestehen. Dabei darf der Name nicht mit zwei Unterstrichen, einem Unterstrich und einem Großbuchstaben, oder einer Ziffer beginnen. In C dient der Funktionsname zugleich als Signatur und ist folglich eindeutig. Es kann also keine Überladungen wie zum Beispiel in Java geben. Folgendes Codebeispiel ließe sich also nicht compilieren:

[1] Für eine unübersichtliche Darstellung sei folgender Link empfohlen:
http://de.wikipedia.org/wiki/Unterprogramm#mediaviewer/Datei:Unterprogramm_Prinzip.png

```
1  static double flaeche( double seiteA )
2  {
3      // Quadrat
4      return seiteA * seiteA;
5  }
6  static double flaeche( double seiteA, double seiteB )
7  {
8      // Rechteck
9      return seiteA * seiteB;
10 }
```

Hinweis: Laut Definition muss der Compiler mindestens die ersten 31 Zeichen eines Funktionsnamens unterscheiden können. Verwendet man Funktionsnamen, bei denen die ersten 31 Zeichen identisch sind, so ist sicherzustellen, dass der Compiler diese als verschiede Funktionen betrachtet. Der GCC-Compiler tut dies!

4. Bei jeder Funktion können mehrere Parameter vorausgesetzt werden, welche durch Kommata voneinander getrennt werden. Parameter werden wie Variablen deklariert - erst der Typ und dann der Name.
   Bei den Parametern unterscheidet man zwischen zwei Varianten. Die Parameter in der Deklaration werden als formale Parameter bezeichnet, an der Aufrufstelle spricht man von aktuellen Parametern.

5. Die Implementation (auch Definition) ist das was die Funktion tut. Sie befindet sich zwischen den beiden äußersten geschweiften Klammern der Funktion, welche sie zugleich begrenzt. In sie werden Anweisungen geschrieben, also z.B. Deklarationen, Berechnungen, ...

2.2 Schlüsselwörter

Achtung: die folgenden Beschreibungen für die Schlüsselwörter beziehen sich auf Funktionen. Bei Variablen sind diese, wenn existent, anders belegt!

- **extern** ist bei Funktionen implizit gegeben und wird dadurch in der Regel nicht mit angegeben. Es sorgt dafür, dass die Funktion global sichtbar ist und auch von außerhalb des Moduls genutzt werden kann (vorausgesetzt der Prototyp ist auch in der Header-Datei deklariert - hierzu später mehr).

- **static** sorgt dafür, dass die Funktion nur innerhalb ihres Moduls aufgerufen werden kann. Durch Funktionspointer jedoch kann man dieses Limit dezent umgehen. Zudem erlaubt *static* dem Compiler implizit zu Inlinen (siehe nächster Punkt), da der Compiler hierdurch mehr Informationen über den Sichtbarkeitsbereich der Funktion besitzt.

  Hinweis: static überschreibt sinngemäß das implizite *extern*. Folglich existieren keine *'extern static'*- oder *'static extern'*-Funktionen.

- **inline** gibt dem Compiler die Anweisung, die Implementation der Funktion an der Position, wo diese aufgerufen wird, direkt einzufügen. Dadurch werden die Sprünge in die Funktion und aus der Funktion zurück nicht mehr

benötigt. Durch inlinen wird die ausführbare Datei am Ende aber oft größer. Inlinen ist also für kleinere Funktionen geeignet.
Der Compiler kann den Befehl zum Inlinen allerdings ignorieren - der Programmierer kann das Inlinen also nur 'empfehlen'.

2.3 Parameterübergabe

Es existieren folgende zwei Möglichkeiten Parameter an Funktionen zu übergeben:

- Bei **call-by-value** wird der Parameter komplett kopiert. Dies ist bei großen Datenmengen sehr ineffizient. Trotzdem ist diese Umsetzung der Standard in C. Falls ein Parameter geändert wird, der mit call-by-value übergeben wurde, hat dieser keinen Einfluss auf den ursprünglichen, aktuellen Parameter.

  Hinweis: Structs und andere Datentypen werden kopiert, während Arrays nicht kopiert werden - sie werden standardmäßig als Pointer übergeben (siehe: call-by-reference).

- Bei **call-by-reference** wird ein Pointer an die Funktion übergeben. Da die Adresse einer Funktion oder Variablen kopiert wird, kann man diese Art der Übergabe auch als call-by-value mit Pointern betrachten.
  Wenn hier Änderungen an den Parametern gemacht werden, sind sie auch nach dem Beenden der Funktion noch vorhanden. Wenn man große Parameter übergibt, lohnt es sich call-by-reference zu nutzen, da dies schneller ist. Es ist allerdings zu beachten, dass der ursprüngliche, aktuelle Parameter veränderbar ist!

3 Die Main-Funktion

Die Main-Funktion ist der Einstiegspunkt in das Programm, wofür sich die nächsten vier Funktionsprototypen anbieten:

```
1 int main( );
2 int main( void );
3 int main( int argc, char **argv );
4 int main( int argc, char *argv[] );
```

Die ersten beiden Prototypen bieten keine Informationen über die beim Programmstart übergebenen Parameter. Da *'char **argv'* und *'char *argv[]'* identisch sind, unterscheiden sich diese Prototypen nur in ihrer Schreibweise. Die beiden letzteren enthalten im Parameter *argc* die Anzahl der übergeben Parameter beim Programmstart, das Array *argv* beinhaltet selbige. *'argv[0]'* beinhaltet stets den Namen des Programms.
Die Main-Funktion ist die einzige Funktion in C, die implizit den Wert 0 zurückliefert. Alternativ kann man die Funktion mit *'return 0;'* verlassen und das Programm erfolgreich beenden. Alternativ ist auch eine Rückgabe von *'EXIT_SUCCESS'* oder *'EXIT_FAILURE'* möglich, welche in der *'stdlib.h'* deklariert sind.

4 Rekursive Funktionen

Rekursive Funktionen rufen sich selber auf nach dem Schema g(g(g(g(...)))). Damit man nicht in eine Dauerschleife gelangt und evtl. ein Stack-Overflow geworfen wird, muss die Funktion erreichbare Abbruchbedingungen enthalten. Man muss also dafür sorgen, dass die Schachtelungstiefe nicht zu groß wird.
Neben der Rekursion existiert noch die indirekte Rekursion. Dabei wird rekursiv abwechselnd zwischen zwei Funktionen gewechselt nach dem Schema: f(g(f(g(...)))).

Wichtig: Jede iterative Funktion kann man in eine rekursive Funktion umwandeln und umgekehrt.

Rekursive Funktionen eignen sich gut für das Durchlaufen von Bäumen z.B. einer Ordnerstruktur. Ein weiteres beliebtes (aber ineffizientes!) Beispiel für Rekursion ist die Berechnung der Fakultät:

```
1  int fakultaet( int x )
2  {
3      if( x >= 0 )
4      {
5          if( x > 1 )
6          {
7              // Rekursiver Aufruf:
8              return x * fakultaet( x - 1 );
9          }
10         return 1;
11     }
12
13     // Fuer fehlerhafte Eingaben:
14     return -1;
15 }
```

5 Prototypen und Header-Dateien

Oft wird beim Programmieren auf Funktionen zugegriffen, die der Reihenfolge im Code nach noch nicht definiert oder deklariert sind. Der Compiler nimmt bei einer unbekannten Funktion an, dass diese später implementiert wird und nimmt einen Integer als Rückgabewert an. Ein Prototyp wird außerhalb von einer Funktion deklariert und teilt dem Compiler den Rückgabewert mit. Die Funktion kann nun ohne die Definition 'aufgerufen' werden, da sie dem Compiler bekannt ist. Es werden also Deklaration und Implementation voneinander getrennt und die Deklaration wird vorrangestellt. Da dem Compiler durch Prototypen mehr Informationen zur Verfügung stehen, kann dieser auch mehr Fehler entdecken.
Bei einem Prototypen entfallen die vorangestellten Schlüsselwörter *extern*, *static* oder *inline*. Zudem müssen keine Parameternamen angegeben werden. Abgeschlossen wird der Prototyp wie üblich mit dem Semikolon, wo normalerweise die Definition folgen würde. Man betrachte folgendes Beispiel:

```
1  // Prototyp
2  double mul( double, double );
3
4  // Die Funktion 'mul' kann hier ohne Probleme bereits
```

```
5  // aufgerufen werden, da sie bereits bekannt ist.
6
7  // Implementation bzw. Definition
8  double mul( double x, double y )
9  {
10     return x * y;
11 }
```

Die Prototypen entfalten erst ihre gesamte Wirkung in Zusammenarbeit mit Header-Dateien. Sie beinhalten keinen ausführbaren Code sondern nur Deklarationen bzw. Prototypen. Diese Header-Dateien kann man mit Hilfe von #include einbinden, wobei der Inhalt der angegebenen Datei vom Präprozessor direkt in den Code eingefügt wird:

```
1  // Einbinden von Librarys aus dem Bibliotheksverzeichnis
2  #include <file.h>
3
4  // Einbinden von Dateien aus dem aktuellen Projektverzeichnis
5  #include "file.h"
```

Durch die Header-Dateien wird auch das Modulkonzept ermöglicht, welches für einen übersichtlicheren Code und weniger Compiler-Aufwand sorgt. Man importiert in der Regel in jedes Modul die benötigten Header-Dateien und kann somit die darin enthaltenen Funktionen benutzen. Das doppelte Einbinden von Header-Dateien kann mit Hilfe eines Include-Guards verhindert werden. Dieser ist wie folgt aufgebaut:

```
1  #ifndef MEIN_HEADER
2  #define MEIN_HEADER
3
4  // Wenn MEIN_HEADER noch nicht definiert ist, so wird dieser
5  // definiert und der Code bis #endif wird ausgegeben. Beim
6  // naechsten Durchlauf wird das Einbinden uebersprungen, da
7  // MEIN_HEADER bereits definiert ist. MEIN_HEADER ist in
8  // diesem Fall ein Platzhalter.
9
10 // Prototypen, Konstanten, ...
11
12 #endif
```

Hinweis: Der Include-Guard wird auch in den Standard-Librarys genutzt, welche sich bei Linux unter /usr/include/ befinden. Eine übersichtliche Header-Datei wäre zum Beispiel 'krandom.h' (C++).

6 Funktionspointer

Wenn man Funktionen genauer betrachtet, handelt es sich bei ihnen nur um Code im Speicher. Ein Funktionsname ist für den Compiler also nicht mehr als ein Pointer auf diesen Code - dieses Prinzip nutzen die Funktionspointer. In der Variable mit dem Funktionspointer wird nur auf die Funktion gezeigt. Zum Zeitpunkt des Aufrufs ist nicht bekannt, welche Funktion sich hinter dem Pointer verbirgt. Ein Funktionspointer muss denselben Funktionstyp wie die Funktion, die ihm zugewiesen werden soll, besitzen, damit ihm dieser Pointer zugewiesen werden kann.

```c
// Der allgemeine Aufbau eines Funktionspointer sieht wie
// folgt aus:
return_type (*variable_name)(parameter_list);
// Die parameter_list benoetigt die Typen der Parameter, die
// Bezeichnungen sind, wie bei den Prototypen, optional.

// Moeglichkeiten der Zuweisung eines Funktionspointers
variable_name = &global_function; // 'korrekte' Variante
variable_name = global_function;  // 'einfache' Variante

// Funktionsaufruf
(*variable_name)(parameter_list); // 'korrekte' Variante
variable_name(parameter_list);    // 'einfache' Variante

// Beispiel
double (*function) (int);
double fct_name(int) { ... }
function = &fct_name;
(*function)(42);
```

Durch Funktionspointer ist es zum Beispiel möglich Funktionen als Parameter zu übergeben. Funktionen können aber auch Funktionspointer (und damit Funktionen) zurückgeben. Da Funktionspointer dazu neigen den Code unübersichtlich zu machen, kann man den Typ mit Typedef als Symbol speichern. Ein Beispiel dazu befindet sich in -Quelle 1- in der Datei 'funktionspointer.c'. Hier findet man auch weitere Beispiele.

6.1 Der *void*-Pointer

Der *void*-Pointer (auch generischer Zeiger) ist ein Spezialfall bei den Pointern. Er ist zu jedem Datentyp kompatibel und kann in beiden Richtungen in jeden anderen Zeiger umgewandelt werden, ohne dass der Zeiger seine Eigenschaften, wie zum Beispiel die Parameter oder den Rückgabetyp, verliert. Möchte man eine Funktion definieren, die eine unbekannte oder unterschiedliche Funktionen zurückgibt, so empfielt sich in diesem Fall ein *void*-Pointer. Man sollte jedoch darauf achten, diesen nur zu verwenden, wenn es unbedingt sein muss.

7 Quellen

1. http://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2014/cgk-14-gawehn-funktion-livecode.zip
2. http://de.wikipedia.org/wiki/Unterprogramm#mediaviewer/Datei:Unterprogramm_Prinzip.png
3. http://de.wikibooks.org/wiki/C-Programmierung
4. http://www.c-howto.de/tutorial-einfuehrung.html
5. http://openbook.galileocomputing.de/c_von_a_bis_z/index.htm
6. http://manderc.manderby.com/types/functionpointertype/
9. http://de.wikibooks.org/wiki/C%2B%2B-Programmierung
10. http://en.wikipedia.org/wiki/Entry_point#C_and_C.2B.2B