Funktionen

Einführung in die Programmierung

Michael Felderer (QE)

Institut für Informatik, Universität Innsbruck

Funktionen

- Funktionen haben die Aufgabe, Teile eines Programms unter einem eigenen Namen zusammenzufassen.
  - Damit kann man dann diesen Programmteil mit einem Namen aufrufen.
  - Beim Aufruf kann man Parameter mitgeben.
  - Funktionen können auch Ergebnisse zurückliefern.
- Funktionen sind ein Mittel zur Strukturierung eines Programms.
  - Bei einem großen Programm kann man nicht alle Anweisungen in einem Block (`main`) zusammenfassen, da sonst der Überblick verloren geht.
  - Ein Programm sollte mit Hilfe von Funktionen in modulare Teile aufgeteilt werden.
- Funktionen sind ein Mittel zur Wiederverwendung.
  - Eine Funktion kann an mehreren Stellen (mit unterschiedlichen Parametern) aufgerufen werden.
  - Die Funktion selbst ist nur einmal im Programm vorhanden und verkürzt daher den Programmtext.

Prozedurale Programmierung

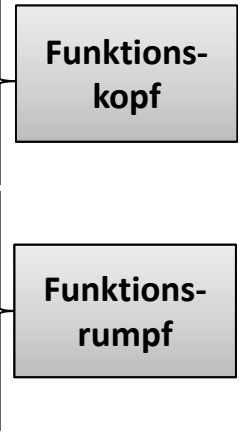
- Zerlegung eins Algorithmus in überschaubare Teile, die anhand einer definierten Schnittstelle aufrufbar sind
- Unterstützung durch das Programmierkonzept der Funktion
- Erweiterung des imperativen Paradigmas
  - Folge von definierten Zustandsübergängen, bei denen festgelegt ist, wie Zustände verändert werden sollen
  - Unterstützung durch Kontrollstrukturen, Variablen und Zuweisungsoperator
- Bei der prozeduralen Programmierung besteht kein Zusammenhalt zwischen Daten und Funktionen
  - In der objektorientierten Programmierung werden Daten und Funktionen in Objekten zusammengefasst

Definition von Funktionen

- Die Definition einer Funktion besteht aus
  - Funktionskopf
  - Funktionsrumpf
- Die Aufgabe einer Funktion ist es, aus Eingabedaten Ausgabedaten zu erzeugen
  - Eingabedaten
    - Per Parameter übergebene Werte
    - Globale Variablen (außerhalb von Funktionen vereinbart)
      - **Sollten nur in Ausnahmefällen verwendet werden!**
  - Ausgabedaten
    - Rückgabewert der Funktion
    - Änderungen an Variablen, deren Adresse an die Funktion über die Parameterliste übergeben wird
      - Wird in der Vorlesung über Zeiger noch genauer besprochen.
    - Änderungen an globalen Variablen (**sollten nur in Ausnahmefällen vorkommen**)

Syntax

```
Spezifizierer Rueckgabetyp Funktionsname (typ_1 formaler_parameter_1,
                                          typ_2 formaler_parameter_2,
                                          …
                                          typ_n formaler_parameter_n)
{
        …
        …
}
```



- Der Funktionskopf beschreibt, wie eine Funktion aufgerufen werden kann (also die Aufrufschnittstelle).
- Der Funktionsrumpf enthält die Anweisungen der Funktion.
- Der Funktionsname muss eindeutig sein!
- Der Spezifizierer (Speicherklassen) ist optional.
- Die Parameter sind auch optional.

Funktionsaufruf

- Eine Funktion wird einmal definiert und kann dann mehrmals aufgerufen werden.
- Wird kein Argument einer Funktion `funktion` übergeben:

```
funktion();
```

- Werden Argumente an eine Funktion `funktion` übergeben:

```
funktion(argument_1,argument_2,..,argument_n);
```

Beispiel (ohne Parameter, ohne Rückgabewert)

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

void hello(void) {
    printf("In der Funktion\n");
}

int main(void) {
    printf("Vor der Funktion\n");
    hello();
    hello();
    printf("Nach der Funktion\n");
    return EXIT_SUCCESS;
}
```

Ausgabe:
Vor der Funktion
In der Funktion
In der Funktion
Nach der Funktion

Parameter

- Die Parameteranzahl wird vom Programmierer festgelegt.
- Werden keine Parameter übergeben, dann folgt nach dem Funktionsnamen das Paar runde Klammern, das in diesem Fall `void` enthält.
- *Variadische* Funktionen haben eine variable Anzahl von Parametern.
  - Solche Parameterlisten werden mit einer Ellipse (drei Punkte am Ende der Parameterliste) konstruiert.
  - Ein Beispiel ist `printf`, weitere Beispiele folgen noch!

Beispiel (2 int-Parameter)

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

void multi(int ival1, int ival2) {
    printf("%d * %d = %d\n", ival1, ival2, ival1 * ival2);
}

int main(void) {
    int val1 = 10, val2 = 20;
    multi(10, 20);
    multi(val1, val2);
    multi(val1 + 10, val2 * 20);
    multi(val1, val2);
    return EXIT_SUCCESS;
}
```

Parameter

Argumente

Ausgabe:

```
10 * 20 = 200
10 * 20 = 200
20 * 400 = 8000
10 * 20 = 200
```

Formaler Parameter und aktueller Parameter

- Verschiedene Bezeichnungen
  - Formaler Parameter oder Parameter
  - Aktueller Parameter oder Argument
- Beim Aufruf einer Funktion mit aktuellen Parametern finden Zuweisungen statt.
  - Ein formaler Parameter wird als lokale Variable (nur in der Funktion verwendbar) angelegt und mit dem Wert des entsprechenden aktuellen Parameters initialisiert (dies wird auch als **call-by-value** bezeichnet).
  - Der aktuelle Parameter kann auch ein beliebiger Ausdruck sein.
- Implizite Typkonvertierung
  - Der Typ des aktuellen Parameters kann sich vom Typ des formalen Parameters unterscheiden.
  - Beim Aufruf wird die **implizite Typkonvertierung** durchgeführt.
  - Achtung: bei variadischen Argumenten wird `float` immer automatisch in `double` umgewandelt!

Rücksprung aus einer Funktion

- Die `return`-Anweisung beendet den Funktionsaufruf.
- Das Programm kehrt zu der Anweisung, in der die Funktion aufgerufen wurde, zurück und beendet diese Anweisung.
- Anschließend wird die nächste Anweisung nach dem Funktionsaufruf abgearbeitet.
- Hat eine Funktion keinen Rückgabewert, dann ist der Rückgabetyp `void`.
  - Bei älteren Versionen von C wird implizit `int` angenommen, wenn kein Rückgabetyp angegeben wird!
    - In C99 gibt es eine Warnung, wenn der Typ weggelassen wird!
- Nach `return` kann ein beliebiger Ausdruck stehen.
  - Wenn der Typ des Ausdrucks nicht mit dem Rückgabetyp übereinstimmt, führt der Compiler eine implizite Typumwandlung durch.
  - `return` kann auch alleine verwendet werden – dann wird die Funktion verlassen, ohne einen Wert zurückzugeben.

Beispiel (Funktionen, Rückgabewert)

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

int add(int a, int b) {
    int sum;
    sum = a + b;
    return sum;
}

int main(void) {
    int x, y, s;
    s = add(10, 20);
    printf("%d\n", s);
    x = s + 10;
    y = s + s;
    s = add(x, y);
    printf("%d\n", s);
    return EXIT_SUCCESS;
}
```

Funktion **add** mit zwei Parametern **a** und **b** vom Typ **int** und Rückgabetyp **int**.

Die lokale Variable **sum** ist nur in **add** sichtbar!

Rückgabe des Inhalts von **sum**.

Aufruf der Funktion **add** mit den Argumenten 10 und 20, der Rückgabewert wird **s** zugewiesen.

Aufruf der Funktion **add** mit den Argumenten **x** und **y**, der Rückgabewert wird **s** zugewiesen.

Ausgabe:

```
30
100
```

Beispiel (Funktionen, Rückgabewert – alternativ)

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

int add(int a, int b) {
    return a + b;
}

int main(void) {
    int x, y, s;
    s = add(10, 20);
    printf("%d\n", s);
    x = s + 10;
    y = s + s;
    s = add(x, y);
    printf("%d\n", s);
    return EXIT_SUCCESS;
}
```

Funktion **add** jetzt mit minimaler Implementierung – hier zulässig

main ist auch eine Funktion mit Rückgabetyp **int**! In diesem Fall wird **main** nichts übergeben!

Und das sind auch Funktionsaufrufe einer Funktion, die uns schon zur Verfügung gestellt wurde (aus **stdio.h**)!

Ausgabe:

```
30
100
```

Interaktive Aufgabe

- Was wurde bei diesem Programm falsch gemacht?

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

float volume_rect(float l, float b, float h) {

  float volume = l*b*h;
}

int main(void) {

  float v = volume_rect(10,10,2);
  printf("Volumen: %f\n", v);

  return EXIT_SUCCESS;
}
```

Rückgabewert

- Kann (muss aber nicht) abgeholt werden.
  - Rückgabewert wird einer Variable zugewiesen.
  - Rückgabewert wird in einem Ausdruck verwendet.
- Funktionsaufruf kann auch nur als reine Ausdrucksanweisung (durch Anhängen eines Strichpunkts) vorkommen.
  - Funktionen, die keinen Rückgabewert (`void`) haben, können nur als Ausdrucksanweisung angeschrieben werden.
- `main`-Funktion
  - Der Rückgabewert der `main`-Funktion wird an das Betriebssystem zurückgegeben.
  - Bei der `main`-Funktion ist die Beendigung mit `exit()` gleichwertig mit einer `return`-Anweisung innerhalb der main-Funktion.
  - Makros `EXIT_SUCCESS` bzw. `EXIT_FAILURE` aus `stdlib.h` geben für bestimmest System erforderlichen Wert für erfolgreiche bzw. nicht erfolgreiche Beendigung zurück

Beispiel (Aufrufe und implizite Typkonvertierung)

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

int max(int a, int b) {
     if (a > b)
          return a;
     else
          return b;
}

int main(void) {
     int s;
     float x;
     s = max(1, 2);
     printf("%d\n", s);
     s = max(6 + 2 * 3, s * 7 + 3);
     printf("%d\n", s);
     x = max(1.8, 2.9);
     printf("%f\n", x);
     printf("%d\n", max(10, max(5, 11)));
     max(3, 4);
     printf("%d\n", printf("  "));
     return EXIT_SUCCESS;
}
```

Ausgabe:

```
2
17
2.000000
11
  2
```

Implizite Typkonvertierung

Verschachtelung von Aufrufen!

Möglich, aber nicht sinnvoll! Macht nur Sinn, wenn die Funktion einen Nebeneffekt (z.B. Ausgabe) hat (siehe **printf**)!

Interaktive Aufgabe

- Warum lässt sich das folgende Programm nicht übersetzen?

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

int main(void) {

  float fval = square(2.5);
  printf("%.2f\n", fval);
  return EXIT_SUCCESS;
}

float square(float a) {

  return a*a;
}
```

Deklaration von Funktionen

- Der Compiler überprüft immer die Konsistenz zwischen Funktionskopf und Funktionsaufruf.
  - Daher muss beim Aufruf der Funktion die Schnittstelle der Funktion (der Funktionskopf) bekannt sein.
- Steht eine Funktion im Programmcode erst nach ihrem Aufruf, so muss eine Vorwärtsdeklaration der Funktion erfolgen.
  - Manchmal möchte man die `main`-Funktion am Anfang der Datei haben.
  - Die restliche Funktionen folgen danach.
- Vorwärtsdeklaration
  - Damit wird dem Compiler der Name der Funktion, der Typ des Rückgabewerts und der Aufbau der Parameterliste bekannt gemacht.
  - Diese Deklaration wird als **Funktionsprototyp** bezeichnet.
    - Die Namen der formalen Parameter müssen nicht mit den Namen der Parameter des Funktionskopfes übereinstimmen.
    - Die Namen der formalen Parameter können weggelassen werden, die Parametertypen müssen aber exakt übereinstimmen!

Beispiel (Vorwärtsdeklaration)

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

int max(int a, int b);
int min(int a, int b);
int sign(int a);

int main(void){
	int a = 10, b = 20, c = 0, d = -10;
	printf("%d\n", max(a, b));
	printf("%d\n", min(a, b));
	printf("%d\n", sign(a));
	printf("%d\n", sign(c));
	printf("%d\n", sign(d));
	return EXIT_SUCCESS;
}

int max(int a, int b){
	return a > b ? a : b;
}

int min(int a, int b){
	return a < b ? a : b;
}

int sign(int a) {
	return a > 0 ? 1 : a < 0 ? -1 : 0;
}
```

Funktionsprototypen.
An dieser Stelle könnte man zum Beispiel auch schreiben:

```c
int max(int, int);
int min(int, int);
int sign(int);
```

Ausgabe:

```
20
10
1
0
-1
```

Behandlung von Bibliotheksfunktionen

- Regeln für die Vorwärtsdeklaration gelten auch für Bibliotheksfunktionen.
- Will man Bibliotheksfunktionen verwenden, so müssen ihre Prototypen bekannt sein.
- Diese befinden sich (neben Makros und Konstanten) in den Header-Dateien.
- Durch das Einbinden der Header-Dateien, werden die Funktionsprototypen der Bibliotheksfunktionen eingefügt!
- Zum Beispiel erzeugt `#include <stdio.h>` folgendes:
  - Der Präprozessor ersetzt an dieser Stelle die Direktive durch den Code in `stdio.h`.
  - In dieser Datei stehen unter anderem die Prototypen von Funktionen (z.B. `printf`).
  - Dadurch kennt der Compiler die Prototypen während der Übersetzung der nachfolgenden Zeilen des eigentlichen Codes.

Gültigkeitsbereiche von Namen

- Der Compiler übersetzt dateiweise.
- In einer Datei gibt es vier Gültigkeitsbereiche:
  - Datei
  - Funktion
  - Block
  - Funktionsprototyp
- Innerhalb einer Datei gelten folgende Regeln:
  - Namen, die in Blöcken eingeführt werden, verlieren am Blockende ihre Bedeutung.
  - Namen der formalen Parameter von Funktionen gelten nur innerhalb der entsprechenden Funktion.
  - Namen externer Variablen sind ab ihrer Deklaration (extern-Deklaration wird noch besprochen) bis zum Ende der Datei gültig.

Globale Variablen

- Globale Variablen können in allen Funktionen einer Datei verwendet werden.
- Gibt es eine lokale Variable mit gleichem Bezeichner (Name), dann wird die lokale Variable verwendet!
- Grundlegende Regel: **Variablen möglichst lokal definieren!**
- Globale Variablen werden **automatisch** initialisiert:

Datentyp	**Initialisierung**
`short, int, long`	0
`char`	'\0'
`float, double`	0.0
Zeiger (werden noch besprochen)	NULL

  - Frage: Warum werden globale Variablen automatisch initialisiert, lokale i.d.R. aber nicht?

Beispiel (Gültigkeitsbereiche – erstes kleines Beispiel)

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

float b; // Globale Variable, überall sichtbar, automatisch initialisiert!

void test1(int a, double b) {
	int c = 10;
	printf("In test1: a=%d, b=%f, c=%d, a*c=%d\n", a, b, c, a * c);
	a = 80;
	b = 30.0;
	printf("In test1: a=%d, b=%f, c=%d, a*c=%d\n", a, b, c, a * c);
}

int main(void) {
	int a = 10;
	int c = 50;
	test1(a, b);
	printf("In main: a=%d, b=%f, c=%d, a*c=%d\n", a, b, c, a * c);
	while (a > 0) {
		int c = 200;
		printf("In main: a=%d, b=%f, c=%d, a*c=%d\n", a, b, c, a * c);
		a--;
	}
	printf("In main: a=%d, b=%f, c=%d, a*c=%d\n", a, b, c, a * c);
	return EXIT_SUCCESS;
}
```

Beispiel (Ausgabe)

```
In test1: a=10, b=0.000000, c=10, a*c=100
In test1: a=80, b=30.000000, c=10, a*c=800
In main: a=10, b=0.000000, c=50, a*c=500
In main: a=10, b=0.000000, c=200, a*c=2000
In main: a=9, b=0.000000, c=200, a*c=1800
In main: a=8, b=0.000000, c=200, a*c=1600
In main: a=7, b=0.000000, c=200, a*c=1400
In main: a=6, b=0.000000, c=200, a*c=1200
In main: a=5, b=0.000000, c=200, a*c=1000
In main: a=4, b=0.000000, c=200, a*c=800
In main: a=3, b=0.000000, c=200, a*c=600
In main: a=2, b=0.000000, c=200, a*c=400
In main: a=1, b=0.000000, c=200, a*c=200
In main: a=0, b=0.000000, c=50, a*c=0
```

Rekursion

- Rekursion liegt dann vor, wenn eine Funktion, ein Algorithmus, eine Datenstruktur, ein Begriff, etc. durch sich selbst definiert wird.
- Auf Funktionen bezogen
  - Eine Funktion heißt **rekursiv**, wenn sie Abschnitte enthält, die wiederum die Funktion direkt oder indirekt aufrufen.
  - Eine Funktion heißt **iterativ**, wenn bestimmte Abschnitte der Funktion innerhalb einer einzigen Ausführung der Funktion mehrfach durchlaufen werden.

Rekursion (ein einfaches Beispiel)

- Berechnung der Summe von *n* Zahlen
- Iterativ ist die Summe definiert durch:
  - sum(n) = 0 + 1 + 2 + … + n
- Rekursiv ist die Summe definiert durch:

$$sum(n) = \begin{cases} 0 & \text{falls n} = 0 \quad \text{Rekursionsanfang} \\ sum(n-1) + n & \text{sonst} \quad \text{Rekursionsschritt} \end{cases}$$

Beispiel (Summe – rekursiv und iterativ)

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

unsigned int sum_iterative(unsigned int num) {
     int sum = 0;
     for (int i = 1; i <= num; i++)
          sum += i;
     return sum;
}

unsigned int sum_recursive(unsigned int num) {
     if (num > 0)
          return num + sum_recursive(num - 1);
     else
          return 0;
}

int main(void){
     printf("%u\n", sum_iterative(10));
     printf("%u\n", sum_recursive(10));
     printf("%u\n", sum_iterative(100));
     printf("%u\n", sum_recursive(100));
     return EXIT_SUCCESS;
}
```

Ausgabe:

```
55
55
5050
5050
```

Interaktive Aufgabe

- Wandeln Sie die folgende iterative Funktionsdefinition in eine rekursive Funktionsdefinition um!

```
unsigned int facul_iterative(unsigned int num) {
  int fac = 1;
  for (int i = 1; i <= num; i++)
    fac *= i;
  return fac;
}
```

Beispiel (Summe – rekursiv und endrekursiv)

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

unsigned int sum_recursive(unsigned int num) {
	if (num > 0)
		return num + sum_recursive(num - 1);
	else
		return 0;
}

unsigned int add_sum(unsigned int m, unsigned int n){
	if (n > 0)
		return add_sum(m + n, n - 1);
	else
		return m;
}

unsigned int sum_recursive2(unsigned int num){
	return add_sum(0, num);
}

int main(void){
	printf("%u\n", sum_recursive(10));
	printf("%u\n", sum_recursive(100));
	printf("%u\n", sum_recursive2(10));
	printf("%u\n", sum_recursive2(100));
	return EXIT_SUCCESS;
}
```

Endrekursion

Endrekursion:
Der letzte Funktionsaufruf ist auch der letzte Schritt in der Berechnung, d.h. mit dem letzten Funktionsaufruf steht das Ergebnis fest.

Ausgabe:

```
55
5050
55
5050
```

Interaktive Aufgabe

- Wandeln Sie die folgende rekursive Funktionsdefinition in eine endrekursive Funktionsdefinition um!

```
unsigned int facul_recursive(unsigned int num) {
  if (num > 0)
    return num * facul_recursive(num - 1);
  else
    return 1;
}
```

Rekursion – Endrekursion

Rekursion sum_recursive(3)	*Endrekursion sum_recursive2(3)*

`sum_recursive(3) = 3 + sum_recursive(2)`
`  sum_recursive(2) = 2 + sum_recursive(1)`
`  sum_recursive(1) = 1 + sum_recursive(0)`
`  sum_recursive(0) = 0`
`  sum_recursive(1) = 1 + 0 = 1`
`  sum_recursive(2) = 2 + 1 = 3`
`sum_recursive(3) = 3 + 3 = 6`
	`sum_recursive2(3) = add_sum(0, 3)`
	`= add_sum(3, 2)`
	`= add_sum(5, 1)`
	`= add_sum(6, 0)`
	`= 6`

Ab hier wird nur mehr **`return`** ausgeführt und zur aufrufenden Funktion zurückgesprungen (wo wieder **`return`** ausgeführt wird usw.)

Vergleich von Rekursion und Iteration

- Iteration und Rekursion sind äquivalent, weil man jede Iteration in eine Rekursion umformen kann und umgekehrt.
- Sehr oft bietet sich aber aufgrund der Problemformulierung eine iterative oder eine rekursive Lösung an.
  - Sehr oft ist die rekursive Lösung viel kürzer.
- Rekursion kann aber schneller zu Speicherproblemen führen!
- Beispiel (zid-gpl)
  - `sum_iterative(300000)` funktioniert (aber mit Overflow!)
  - `sum_recursive(300000)` führt zu einem „Absturz“
    - Problem: Stack-Overflow durch zu viele Funktionsaufrufe
- Endrekursion kann von einem Compiler automatisch in eine iterative Form umgewandelt werden.

Rekursion – Stack

- Wenn eine Funktion aufgerufen wird, dann müssen entsprechende Daten für die Funktion abgelegt werden.
- Die Daten der aufrufenden Funktion (z.B. `main`-Funktion) müssen in der Zwischenzeit aber auch irgendwo gespeichert werden.
  - Nach der Rückkehr aus dem Funktionsaufruf wird in der aufrufenden Funktion mit den vorhandenen Daten weitergearbeitet!
- Es existiert der sogenannte Stack, wo bei Bedarf vom Programm Speicher reserviert und wieder freigegeben wird.
- Für **jeden neuen Funktionsaufruf** wird ein Datenblock (Stack-Frame) angelegt.
  - In diesem Datenblock werden die formalen Parameter, die lokalen Variablen und die Rücksprungadresse zur aufrufenden Funktion gespeichert.
  - Wird die Funktion beendet (`return`-Anweisung oder Ende des Anweisungsblocks), dann werden diese Daten wieder freigegeben.
    - **Alle lokalen Variablen sind damit weg!**

Rekursion – Stack Overflow

- Was passiert bei der Rekursion?
  - Funktion ruft Funktion auf, die wieder Funktion aufruft, die wiederum …..
  - Es werden immer mehr Datenblöcke angelegt und irgendwann ist der Speicher im Stack aufgebraucht!
  - Neue Datenblöcke können nicht mehr angelegt werden und daher bricht das Programm ab.

Verschachtelte Funktionen?

- In C (C-Standard) gibt es **nicht** die Möglichkeit Funktionen zu verschachteln, d.h. eine Funktion in einer weiteren Funktion zu definieren!
  - Nicht in Standard-C möglich:

```
void x(){
        void y(){
                …
        }
        …
}
```

- Das bedeutet aber nicht, dass kein Compiler dies unterstützt.
  - Für die gcc-Unterstützung siehe:
    - http://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html
  - Sollte in der Übung nicht verwendet werden!

main-Funktion

- In üblichen Ausführungsumgebungen, in denen ein in C erstelltes Programm unter der Kontrolle des Betriebssystems ist, ist der Name der ersten Funktion immer `main`.
- Der Rückgabewert beim Beenden eines Programms ist abhängig von der Umgebung des Betriebssystems.
  - Unter Linux/Unix bedeutet ein Rückgabewert von 0, dass ein Programm erfolgreich beendet wurde; alles andere bedeutet, dass ein Fehler aufgetreten ist.
  - Bei anderen Betriebssystemen kann auch ein anderer Wert eine erfolgreiche Beendigung anzeigen.
  - Deshalb sollte man die entsprechenden Makros `EXIT_SUCCESS` bzw. `EXIT_FAILURE` benutzen, die in der entsprechenden Umgebung auf den richtigen Wert abgebildet werden.
- In C99 muss die `main`-Funktion nicht unbedingt eine `return`-Anweisung enthalten.

Fallgruben

- Die Reihenfolge der Argument-Auswertung ist nicht festgelegt
  - Der C- Standard schreibt **nicht** vor, ob die Argumente von links nach rechts oder von rechts nach links ausgewertet werden.
    - Beispiel mit Warnings: `m(a*2,a++);`
  - Die Argumente und Funktionsbezeichner müssen nur vollständig ausgewertet sein, bevor mit der Ausführung des Funktionscodes begonnen wird.
- Die Aufrufreihenfolge von Funktionen in Ausdrücken ist **nicht** festgelegt.