Modellierung und Programmierung

Dr. Martin Riplinger

14.11.2012

Funktionen

- Funktion $\approx$ Zusammenfassung eines Anweisungsblocks zu einer **aufrufbaren** Einheit
- Gehört zu den wichtigsten Konzepten fast aller Programmiersprachen
- Aufgaben von Funktionen:
  1. Wiederverwendbarkeit von einmal geschriebenem Code
  2. Strukturierung und Vereinfachung von Code
     $\rightsquigarrow$ bessere Übersicht und Lesbarkeit
  3. einfachere Fehlersuche, separates Testen möglich
  4. leichteres Hinzufügen weiterer Funktionalitäten

$\rightsquigarrow$ Die sinnvolle Strukturierung eines Programms in Unterprogramme ist einer der wichtigsten Schritte bei der Programmierung!

Charakterisierung von Funktionen

Eine Funktion

- besitzt einen **sinnvollen** (=sprechenden) Namen, mit dem sie aufgerufen wird.
- nimmt eine (möglicherweise leere) **Liste von Parametern** mit festgelegten Datentypen als Eingabe.
- hat einen Anweisungsblock, der bei ihrem Aufruf ausgeführt wird.
- liefert **nichts** oder **einen Wert** eines festgelegten Datentyps als Ausgabe
- „Mathematische“ Schreibweise:

$$\text{Funktion} : \{\texttt{int}, \texttt{float}, \ldots\}^N \longrightarrow \{\texttt{int}, \texttt{float}, \ldots\} \text{ oder } \{\}, \quad N \geq 0$$

$$(\text{Parameter1}, \ldots, \text{ParameterN}) \underset{\text{Anweisungen}}{\longmapsto} \text{Ausgabewert}$$

- Im Gegensatz zu mathematischen Funktionen kann der Anweisungsblock auch Befehle enthalten, die nicht direkt etwas mit der Ausgabe zu tun haben (auch „Funktion: $\{\} \to \{\}$“ kann in C sinnvoll sein).

Syntax

Deklaration

```
Rückgabetyp Funktionsname(ParTyp1 [Par1], ..., ParTypN [ParN]);
```

Definition

```
Rückgabetyp Funktionsname(ParTyp1 Par1, ..., ParTypN ParN)
{
  Anweisungsblock
}
```

Unterschied zwischen Deklaration und Definition: verschiedene Abstraktionsebenen!

Deklaration (= Signatur)

- legt fest, *was* eine Funktion tut.
- muss *vor dem erstmaligen Aufruf* und *außerhalb von* `main` im Code stehen.
- endet mit einem Semikolon.
- Parameter-Datentypen genügen.

Definition

- legt fest, *wie* eine Funktion etwas tut.
- kann an einer beliebigen Stelle *außerhalb von* `main` im Code stehen.
- hat kein Semikolon am Ende.
- Parameter sind mit Typ und Name anzugeben.

Beispiel

```c
/** Praeprozessordirektive ****/
#include <stdio.h>


/** Funktionsdeklarationen ****/
float summe(float a, float b);     // auch moeglich:
                                   // float summe(float, float);


/** Hauptprogramm ****/
int main(void)
{
  float sum;
  sum = summe(3.5, 1);              // Funktionsaufruf
}


/** Funktionsdefinitionen ****/
float summe(float a, float b)
{
  float sum = a + b;                // Funktionsrumpf
  return sum;                       // Funktionsrumpf
}
```

Der Datentyp `void`

Funktionen ohne Rückgabe:

- Deklaration: `void Funktion(Parameterliste);`
- Beispiel: `void srand48(long seed);`
- Verwendung: z. B. Bildschirmausgabe, Statusänderung eines externen „Mechanismus“
- In anderen Programmiersprachen (z.B. Pascal) oft als *Prozedur* bezeichnet. In C gibt es keine Differenzierung zwischen Funktion und Prozedur.

Funktionen ohne Parameter:

- Deklaration: `Rückgabetyp Funktion(void);`
- Beispiel: `double drand48 (void);`
- Verwendung: z. B. Ausführung von externen „Mechanismen“

Kombination: z. B. `void abort(void);` (Gewaltsames Ende des Programms)

`return` und `main`

`return`

- Mit der Ausführung der `return`-Anweisung wird die **aktuelle** Funktion **sofort** beendet.
- Enthält der Funktionsrumpf keine `return`-Anweisung, so endet die Ausführung des Rumpfes bei Erreichen der letzten schließenden geschweiften Klammer.
  → schlechter Stil!
- Beachte: Es kann immer nur ein (skalarer) Wert zurückgegeben werden.

main

- `main` ist in Wirklichkeit eine Funktion (mit Rückgabetyp `int`).
- Quelltext innerhalb von `main` wird als *Hauptprogramm* bezeichnet.
- Es sind äquivalent: `main()` `int main()` `int main(void)`
- `main` liefert standardmäßig den Rückgabewert `0` wenn keine Fehler aufgetreten sind.
- Mit `return ...;` können andere Werte zurückgeliefert werden.
  → Fehlerbehandlung

Beispiel 1: Funktion mit Rückgabewert

Die charakteristische Funktion eines Intervalls $[a, b]$ mit $a < b$ ist definiert als

$$\chi_{[a,b]}(x) := \begin{cases} 1 & , \quad x \in [a, b] \\ 0 & , \quad \text{sonst.} \end{cases}$$

Implementierung: je nach Wert von x wird 1.0 oder 0.0 zurückgegeben.

Definition im Code:

```
// Charakteristische Funktion des Intervalls [a,b]
// (1 innerhalb, 0 ausserhalb)
float charFunkIntervall(const float a, const float b, float x)
{
  if(a >= b)
  // Hier muss eine Fehlerbehandlung hin

  if ((x >= a) && (x <= b))
  return 1.0;
  else
  return 0.0;
}
```

Beispiel 2: Funktion ohne Rückgabewert

```
void geplapper(int zahl1, double zahl2, char c)
{
  printf("Diese Funktion erzeugt eine Menge (sinnloser) Ausgaben ");
  printf("am Bildschirm.\n\n");

  printf("Jetzt noch eine horizontale Linie, dann geht's los!\n");
  printf("-------------------------------------------------\n");

  printf("Zahl1 = %d, Zahl2 = %f\n", zahl1, zahl2);

  if(c == '+')
  printf("Die Summe der beiden Zahlen ist %f\n\n", zahl1+zahl2);

  printf("So, jetzt bin ich fertig!\n");

  return;
}
```

Mathematische Funktionen

- Nutzung erfordert Präprozessordirektive `#include <math.h>`
- Compilierung
  `gcc ProgrammName.c -o ProgrammName -lm`

Signatur	Bedeutung
`int abs(int a)`	$\|a\|$
`float fabsf(float a)`	$\|a\|$
`double fabs(double a)`	$\|a\|$
`double sqrt(double x)`	$\sqrt{x}$
`double pow(double b, double e)`	b^e
`double exp(double x)`	e^x
`double log(double x)`	$\ln(x)$
`double log10(double x)`	$\log_{10}(x)$

Mathematische Funktionen

Signatur	Bedeutung
`double sin(double x)`	$\sin(x)$
`double cos(double x)`	$\cos(x)$
`double tan(double x)`	$\tan(x)$
`double asin(double x)`	$\arcsin(x)$
`double acos(double x)`	$\arccos(x)$
`double atan(double q)`	$\arctan(q) \in (-\frac{\pi}{2}, \frac{\pi}{2})$
`double atan2(double x, double y)`	$\arctan(y/x) \in (-\pi, \pi]$
`double sinh(double x)`	$\sinh(x)$
`double cosh(double x)`	$\cosh(x)$
`double floor(double x)`	$\lfloor x \rfloor$
`double ceil(double x)`	$\lceil x \rceil$

Konstanten in `math.h`

Name	Bedeutung
`M_E`	e
`M_LOG2E`	$\log_2(e)$
`M_LOG10E`	$\log_{10}(e)$
`M_LN2`	$\ln(2)$
`M_LN10`	$\ln(10)$
`M_PI`	π
`M_PI_2`	$\pi/2$
`M_PI_4`	$\pi/4$
`M_1_PI`	$1/\pi$
`M_SQRT2`	$\sqrt{2}$
`M_SQRT1_2`	$1/\sqrt{2}$

Bemerkung: Die Namen der Konstanten werden vom Präprozessor im Code **textuell** durch die entsprechenden Werte ersetzt, z. B. `M_PI` durch `3.14159265358979323846`. (Gleitkommazahlen werden automatisch als `double` interpretiert.)

Call by Value

Beispiel: Vertauschen zweier Werte (?)

```c
#include <stdio.h>

// Funktionsdeklaration
void vertausche(int p, int q);      // oder (int, int)


// Hauptprogramm
int main(void)
{
  int a = 1, b = 3;
  vertausche(a, b);
  printf("a = %d, b = %d\n", a, b);
  return 0;                         // guter Stil
}

// Funktionsdefinition
void vertausche(int p, int q)
{
  int hilf = p;
  p = q;
  q = hilf;
  return;
}
```

Ausgabe: `a = 1, b = 3`

Die Werte wurden gar nicht vertauscht! Warum?

Call by value

Definition

Bei einem Funktionsaufruf werden nicht die Variablen als solche, sondern lediglich ihre Werte, d.h. Kopien der Variableninhalte übergeben.

+ Funktionsaufrufe können direkt als Parameter für eine andere Funktion verwendet werden, da der Wert und nicht die Funktion selbst übergeben wird.

  Beispiel: `printf("Wurzel von 2 = %f\n", sqrt(2.0));`

+ Unbeabsichtigte Manipulation der Variablen durch Funktionen wird vermieden.

– Der Manipulation von Variablen sind Grenzen gesetzt, da immer nur ein (skalarer) Wert zurückgeliefert werden kann. (Ausweg: *Zeiger* und *Call by reference*, später)

Häufige Fehlerquelle: Annahme, dass Funktionsparameter durch die Funktion verändert werden können. **Dem ist nicht so!**

Beispiel: `float a = 2.0; sqrt(a);`

Scope und Lifetime

Definition (Scope)

Der Scope (Sichtbarkeit) eines deklarierten Objekts (Variable oder Funktion) ist der Bereich im Quelltext, in dem es bekannt, d. h. mit seinem Namen aufrufbar ist.

Generell gilt:

- Variablen sind innerhalb des **textuellen** Codeblocks sichtbar, in dem sie deklariert wurden.
- Funktionen sind ab ihrer Deklaration in der gesamten Datei sichtbar.

Definition (Lifetime)

Die Lifetime (Lebenszeit) einer Variablen beschreibt den Zeitraum, in dem der Speicherbereich der Variablen für sie reserviert ist.

Grundregel:

- Eine Variable existiert vom Moment ihrer Deklaration bis zu dem Zeitpunkt, an dem der Block, welcher die Deklaration umschließt, verlassen wird.

Lokale und globale Variablen

Lokale Variablen

- Lokale Variablen werden zu Beginn eines Anweisungsblocks deklariert.
- Lifetime: Bis zum Ende des Anweisungsblocks, also auch in inneren Blöcken
- Scope: Innerhalb des Blocks, sofern sie nicht durch Variablen gleichen Namens in untergeordneten Blöcken überdeckt werden
- Funktionen liegen *textuell* außerhalb jedes anderen Blocks → lokale Variablen sind dort **generell nicht sichtbar**.

Globale Variablen

- Globale Variablen werden außerhalb aller Funktionen (einschl. `main`) deklariert. Namenskonvention: Unterstrich am Ende des Namens, z. B. `int var_ = 42;`
- Lifetime: Gesamte Dauer der Programmausführung (auch über Dateigrenzen hinweg → später)
- Scope: Überall (auch in Funktionen)
- Gefahren: Namenskonflikte, unkontrollierte Manipulation, Chaos

  → Nutzung globaler Variablen auf ein Minimum reduzieren!

Sichtbarkeit: Beispiel 1

```c
#include <stdio.h>

int main(void)
{
  int a = 4;
  {
    int a = 5;
    printf("Innen: a = %d\n", a);
  }
  printf("Aussen: a = %d\n", a);

  return 0;
}
```

Ausgabe:

```
Innen: a = 5
Aussen: a = 4
```

Sichtbarkeit: Beispiel 2

```c
#include <stdio.h>

int main(void)
{
  int i;

  for (i=5; i<10; i++)
  {
    int i=0;
    i++;
    printf("In Schleife: i = %2d\n", i);
  }

  printf("Nach Schleife: i = %2d\n", i);
  return 0;
}
```

Ausgabe:

```
In Schleife:   i =
In Schleife:   i =
In Schleife:   i =
In Schleife:   i =
In Schleife:   i =
Nach Schleife: i =
```

Lokale und globale Variablen

```
1  #include <stdio.h>
2
3  int a_ = 10;          // globale Variable
4
5  int funktion(int);
6  void prozedur(void);
7
8  int main(void)
9  {
10   prozedur();
11   prozedur();
12   funktion(a_);
13   printf("a_ = %d\n", a_);
14   return 0;
15 }
16
17 void prozedur(void)
18 {
19   a_ *= a_;
20   return;
21 }
22
23 int funktion(int d_)        // schlechter Stil: nur globale Variablen
24 {                           // sollten mit "_" enden!
25   return(--d_);
26 }
```

Ausgabe: `a_ =`

Automatische und statische Variablen

Bisher waren alle Variablen automatische Variablen, d.h. sie existieren bis zu dem Zeitpunkt, an dem der Block, welcher die Deklaration umschließt, verlassen wird.

Statische Variablen

- Möglichkeit, dass eine Funktion beim nächsten Durchlauf die Information, die in der Variablen gespeichert wurde, verwenden kann (wie in einem Gedächtnis)
- Sichtbarkeit: Innerhalb des Blocks, sofern sie nicht durch Variablen gleichen Namens in untergeordneten Blöcken überdeckt werden
- Deklaration: `static Datentyp Name=Wert;`

Beispiel:

```
1 void zaehle() {
2    static int i = 1; // i wird (nur) beim ersten Aufruf von zahle initialisiert
3    printf("%d\n", i);
4    i = i + 1;
5 }
```

Rekursive Programmierung

Rekursion
Aufruf einer Funktion durch sich selbst.

Iteration
Wiederholung eines Anweisungsblocks.

Bemerkungen zur rekursiven Programmierung:

- Man muss die *Schachtelungstiefe* der Rekursion selbst überwachen, sonst kann es zum sog. *Stapelüberlauf* (engl. *stack overflow*) kommen.
- Meist sind Iteration und Rekursion äquivalent, aber häufig ist die Überführung in die jeweils andere Variante nicht offensichtlich.
- Je nach Fragestellung (Laufzeit-, Speicher-, Lesbarkeitsoptimierung) entscheidet man sich für eine der beiden Methoden.
- Viele effiziente Sortieralgorithmen oder *Divide-and-Conquer*-Techniken basieren auf dem Prinzip der Rekursion.

Rekursive Programmierung

Einfaches Beispiel: Fakultät $n! = n(n-1)\cdots 1 = \text{fac(n)}$

Es gilt: $$\text{fac(n)} = \begin{cases} \text{n * fac(n-1)} & , \text{n} > 1 \\ 1 & , \text{n} \in \{0, 1\} \end{cases}$$

Interessanteres Beispiel: Quersumme einer natürlichen Zahl

Algorithmus

1. Die Quersumme einer einstelligen Zahl ist die Zahl selbst.
2. Die Quersumme einer mehrstelligen Zahl ist die Summe der letzten Ziffer und der Quersumme der Zahl ohne ihre letzte Ziffer.

Beispiel:

- Quersumme(5) = 5
- Quersumme (31415) = Quersumme (3141) + 5

Quersumme iterativ

```c
#include <stdio.h>

int main(void)
{
  int zahl, qsumme = 0;

  printf("Zahl = ");
  scanf("%d", &zahl);

  printf("Quersumme(%d) = ", zahl);

  while (zahl)
  {
    qsumme += zahl % 10;
    zahl /= 10;
  }

  printf("%d\n", qsumme);

  return 0;
}
```

Quersumme rekursiv

```c
#include <stdio.h>

int qsumme(int);

int main(void)
{
  int zahl;
  printf("Zahl = ");
  scanf("%d", &zahl);

  printf("\nQuersumme(%d) = %d\n",zahl, qsumme(zahl));

  return 0;
}


int qsumme(int zahl)
{
  if (zahl / 10)
    return zahl % 10 + qsumme(zahl / 10);

  return zahl;
}
```