Kapitel 9

Funktionen

Ein C-Programm gliedert sich ausschließlich in Funktionen. Beispiele für Funktionen wurden bereits vorgestellt:

- Die Funktion `main()`, die den Ausführungsbeginn des Programms markiert und somit das Hauptprogramm darstellt.
- Bibliotheksfunktionen, die häufiger benötigte höhere Funktionalität bereitstellen (z.B. `printf()`, `scanf()`, `sqrt()`, `strcpy()` etc).

C verfügt nur über einen sehr kleinen Sprachumfang, stellt jedoch eine Vielzahl an Funktionen in Bibliotheken für fast jeden Bedarf bereit. Was aber, wenn man eine Funktion für eine ganz spezielle Aufgabe benötigt und nichts Brauchbares in den Bibliotheken vorhanden ist? Ganz einfach: Man schreibt sich diese Funktionen selbst.

9.1 Deklaration, Definition und Rückgabewerte

Die Deklaration einer Funktion hat die Gestalt

```
Datentyp Funktionsname(Datentyp1,...,DatentypN);
```

Die Deklaration beginnt mit dem Datentyp des Rückgabewertes, gefolgt vom Funktionsnamen. Es folgt eine Liste der Datentypen der Funktionsparameter. Bei der Deklaration können auch die Namen für die Funktionsparameter vergeben werden:

```
Datentyp Funktionsname(Datentyp1 Variable1,...,DatentypN VariableN);
```

Die Deklaration legt jedoch nur das Allernotwendigste fest, so ist z.B. noch nichts darüber gesagt, was mit den Funktionsparametern im Einzelnen geschieht und wie der Rückgabewert gebildet wird. Diese wichtigen Aspekte werden in der Definition der Funktion behandelt:

```
Datentyp Funktionsname(Datentyp1 Variable1,...,DatentypN VariableN)
{
  Deklaration der Funktionsvariablen
  Anweisungen
}
```

Der Funktionsrumpf besteht ggf. aus Deklarationen von weiteren Variablen (z.B. für Hilfsgrößen oder den Rückgabewert) sowie Anweisungen.

Folgendes ist bei Funktionen zu beachten:

- Die Deklaration bzw. Definition von Funktionen wird außerhalb jeder anderen Funktion - speziell `main()` - vorgenommen.
- Funktionen müssen vor ihrer Verwendung zumindestens deklariert sein. Unterlässt man dies, so nimmt der Compiler eine implizite Deklaration mit Standardrückgabewert int vor, was eine häufige Ursache für Laufzeitfehler darstellt.
- Deklaration/Definition können in beliebiger Reihenfolge erfolgen.
- Deklaration und Definition müssen konsistent sein, d.h. die Datentypen für Rückgabewert und Funktionsparameter müssen übereinstimmen.

Beispiel 9.1 Funktion.

```
#include <stdio.h>

/***********************************/
/* Deklaration der Maximum-Funktion */
/***********************************/
float maximum (float, float);

/*********************************************************/
/* Hauptprogramm                                         */
/*********************************************************/
int main()
{
       float a=3.0,b=2.0;
       printf("Das Maximum von %f und %f ist %f\n",a,b,maximum(a,b));
       return 0;
}

/*************************************************************/
/* Definition der Maximum-Funktion                           */
/*************************************************************/
float maximum (float x, float y)
{
       /* Die Funktion maximum() erstellt Kopien
        * der Funktionswerte (reserviert neuen Speicher)
        * und speichert sie in x bzw. y */
       float maxi;

       if (x>y) maxi=x;
       else maxi=y;

       return maxi;          /* R\"uckgabewert der Funktion */
}
```

□

9.2 Lokale und globale Variablen

Variablen lassen sich nach ihrem Gültigkeitsbereich unterteilen:

- *lokale Variablen:* Sie gelten in dem Anweisungsblock (z.B. Funktionenrumpf oder Schleifenrumpf), in dem sie deklariert wurden. Für diesen Block gelten

sie als lokal. Bei der Ausführung des Programms existieren die Variablen bis zum Verlassen des Anweisungsblocks.

- *globale Variablen:* Sie werden außerhalb aller Funktionen deklariert/definiert (z.B. direkt nach den Präprozessordirektiven) und sind zunächst im gesamten Programm einschließlich aller Funktionen gültig. Dies bedeutet speziell, dass jede Funktion sie verändern kann. (Achtung: unvorhergesehener Programmablauf möglich.)

Variablen die auf einer höheren Ebene (global oder im Rumpf einer aufrufenden Funktion) deklariert/definiert wurden, können durch Deklaration gleichnamiger lokaler Variablen im Rumpf einer aufgerufenen Funktion „verdeckt“ werden. In diesem Zusammenhang spricht man von Gültigkeit bzw. Sichtbarkeit von Variablen.

Beispiel 9.2 Gültigkeitsbereich von Variablen.

```
/*************************************************/
/* Im Beispiel wird 4 x die Variable a deklariert
/*************************************************/
#include <stdio.h>

/***********************************/
/* Deklaration der Funktion summe() */
/***********************************/
int  summe (int, int);

/*********************************************************/
/* Deklaration von globalen Variablen                    */
/*********************************************************/
int a = 1;

/*********************************************************/
/* Hauptprogramm                                         */
/*********************************************************/
int main()
{
    printf("start %d\n",a);
    int a=2;   /* a in main() = 2 und ueberdeckt globales a*/
    printf("1.stelle %d\n",a);
    {
        int a=2; /* lokales a in main() = 2 und ueberdeckt a in main() */
        printf("2.stelle %d\n",a);
        a=a+1;   /* lokales a in main() wird um 1 erh\"oht */
        printf("3.stelle %d\n",a);
        a=summe(a,a); /* lokales a in main() = 2 x lokales a in main()
                       * Gleichzeitig wird das globale a in
                       * der Funktion summe um 1 erh\"oht */
        /* lokales a in main() wird gel\"oscht */
        printf("4.stelle %d\n",a);
    }
    printf("5.stelle %d\n",a);
    a=summe(a,a);      /* a in main = 2 x lokales a in main()
                        * Gleichzeitig wird das globale a in
                        * der Funktion summe um 1 erh\"oht */
    printf("6.stelle %d\n",a);

    return 0;
}

/*************************************************************/
```

```
/* Definition Summen-Funktion */
/*************************************************************/
int summe (int x, int y)
{
   /* Die Funktion summe() erstellt Kopien
    * der Funktionswerte (reserviert neuen Speicher)
    * und speichert sie in x bzw. y */

   printf("summe: 1.stelle %d\n",a);
   a=a+1; /* globales a wird um 1 erh\"oht */
   printf("summe: 2.stelle %d\n",a);

   int a=0;   /* a in der Funktion summe() */
   printf("summe: 3.stelle %d\n",a);
   a=x+y;            /* a in der Funktion summe = x+y */
   printf("summe: 4.stelle %d\n",a);

   return a;  /* a in der Funktion wird zur\"uckgegeben */
              /* a,x,y in der Funktion summe werden gel\"oscht */
}
```

Die Ausgabe ist

```
start 1
1.stelle 2
2.stelle 2
3.stelle 3
summe: 1.stelle 1
summe: 2.stelle 2
summe: 3.stelle 0
summe: 4.stelle 6
4.stelle 6
5.stelle 2
summe: 1.stelle 2
summe: 2.stelle 3
summe: 3.stelle 0
summe: 4.stelle 4
6.stelle 4
```

Dieses Beispiel zeigt, dass man sehr gut aufpassen muss, wenn man den gleichen Namen für unterschiedliche Variablen verwendet. Auch wegen der Übersichtlichkeit empfiehlt es sich, für jede Variable einen anderen Namen zu verwenden. □

9.3 Call by value

Die Standardübergabe von Funktionsparametern geschieht folgendermaßen: An die Funktion werden Kopien der Variablen als Parameter übergeben und von dieser zur Verarbeitung genutzt. Die ursprüngliche Variable bleibt von den in der Funktion vorgenommenen Manipulationen unberührt (es sei denn, sie wird durch den Rückgabewert der Funktion überschrieben).

Beispiel 9.3 Call by value I.

```
#include <stdio.h>

/***********************************/
/* Deklaration der Funktion setze() */
```

```
/************************************/
void setze (int);

/*****************************/
/* Hauptprogramm           */
/*****************************/
int main()
{
        int b=0;
        setze(b);
        printf("b=%i\n",b);
        return 0;
}

/************************************/
/* Definition der Funktion setze () */
/************************************/
void setze (int b)
{
        b=3;
}
```

Die Ausgabe lautet:

```
b=0
```

Die Funktion `setze()` hat nur eine Kopie der Variablen `b` als Parameter erhalten und auf einen neuen Wert gesetzt. Die eigentliche Variable behielt ihren Wert. □

Beispiel 9.4 Call by value II.

```
#include <stdio.h>

/************************************/
/* Deklaration der Funktion setze() */
/************************************/
int setze (int);

/*****************************/
/* Hauptprogramm           */
/*****************************/
int main()
{
        int b=0;
        b=setze(b);
        printf("b=%i\n",b);
        return 0;
}

/************************************/
/* Definition der Funktion setze () */
/************************************/
int setze (int b)
{
        b=3;
        return b;
}
```

Die Ausgabe lautet:

```
b=3
```

Die Funktion `setze()` hat wieder nur eine Kopie der Variablen `b` als Parameter erhalten und auf einen neuen Wert gesetzt. Durch das Zurückliefern und die Zuweisung an die eigentliche Variable `b` wurde die Änderung wirksam. □

9.4 Call by reference

Bei *call by reference* wird der Funktion nicht eine Kopie der Variablen selbst, sondern in Form eines Pointers auf die Variable eine *Kopie der Adresse der Variablen* übergeben. Über die Kenntnis der Variablenadresse kann die Funktion den Variableninhalt manipulieren. Hierzu kommen beim Aufruf der Funktion der Adressoperator und im Funktionsrumpf der Inhaltsoperator in geeigneter Weise zum Einsatz.

Beispiel 9.5 Call by reference.

```
#include <stdio.h>

/************************************/
/* Deklaration der Funktion setze() */
/************************************/
void setze (int *);

/*****************************/
/* Hauptprogramm             */
/*****************************/
int main()
{
       int b=0;
       setze(&b);
       printf("b=%i\n",b);
       return 0;
}

/************************************/
/* Definition der Funktion setze () */
/************************************/
void setze (int *b)
{
       *b=3;
}
```

Die Ausgabe lautet:

b=3

Der Funktion `setze()` wird ein Zeiger auf eine int-Variable übergeben und sie verwendet den Inhaltsoperator `*`, um den Wert der entsprechenden Variablen zu verändern. Im Hauptprogramm wird der Zeiger mit Hilfe des Adressoperators `&` erzeugt. □

9.5 Rekursive Programmierung

Bisher haben Funktionen ihre Aufgabe in einem Durchgang komplett erledigt. Eine Funktion kann ihre Aufgabe aber manchmal auch dadurch erledigen, dass sie sich selbst mehrmals aufruft und jedesmal nur eine Teilaufgabe löst. Das führt zu rekursiven Aufrufen.

Beispiel 9.6 Rekursive Programmierung. Das folgende Programm berechnet x^k, $x \in \mathbb{R}$, $k \in \mathbb{N}$, rekursiv.

```
#include <stdio.h>

/* Deklaration potenz-Funktion */
double potenz (double, int);

/* Hauptprogramm */
int main()
{
        double x;
        int k;
        printf("Zahl x : ");  scanf("%lf",&x);
        printf("Potenz k : "); scanf("%i",&k);
        printf("x^k = %f \n",potenz(x,k));

        return 0;
}

/* Definition potenz-Funktion */
double potenz (double x, int k)
{
        if (k<0) /* Falls k < 0 berechne (1/x)^(-k) */
        {
                return potenz(1.0/x,-k);
        }
        else
        {
                if (k==0) /* Rekursionsende */
                {
                        return 1;
                }
                else
                {
                        return x*potenz(x,k-1); /* Rekursionsaufruf */
                }
        }
}
```

Dieser Rekursion liegt die Darstellung

$$x^k = \underbrace{x(x(x(\ldots 1)))}_{k}$$

zu Grunde. Die Funktion `potenz` ruft sich solange selbst auf, bis der Fall `k==0` eintritt. Das Ergebnis dieses Falls liefert sie an die aufrufende Funktion zurück. □

Achtung ! Bei der rekursiven Programmierung ist stets darauf zu achten, dass der Fall des Rekursionsabbruchs (im Beispiel `k==0`) immer erreicht wird, da sonst die Maschine bis zum nächsten Stromausfall oder bis der Speicher voll ist (mit jedem Funktionsaufruf werden ja lokale Variablen angelegt) rechnet.

9.6 Kommandozeilen–Parameter

Ausführbaren Programmen können beim Aufruf Parameter übergeben werden, indem man nach dem Programmnamen eine Liste der Parameter (Kommandozeilen–Parameter) anfügt. In `C`–Programmen können so der Funktion `main` Parameter übergeben werden. Zur Illustration wird folgendes Beispiel betrachtet.

Beispiel 9.7 Kommandozeilen–Parameter.

```
/* 1  */ # include <stdio.h>
/* 2  */
/* 3  */ int main(int argc, char* argv[])
/* 4  */ {
/* 5  */       int zaehler;
/* 6  */       float zahl,summe=0;
/* 7  */
/* 8  */       for (zaehler=0;zaehler < argc ; zaehler++)
/* 9  */       {
/* 10 */              printf("Parameter %i = %s \n",zaehler,argv[zaehler]);
/* 11 */       }
/* 12 */       printf("\n");
/* 13 */       for (zaehler=1;zaehler < argc ; zaehler++)
/* 14 */       {
/* 15 */              sscanf(argv[zaehler],"%f",&zahl);
/* 16 */
/* 17 */              summe=summe+zahl;
/* 18 */       }
/* 19 */       printf("Die Summe der Kommandozeilen-Parameter : %f\n",summe);
/* 20 */
/* 21 */       return 0;
/* 22 */ }
```

Nach dem Start des obigen Programms zum Beispiel durch

```
a.out 1.4 3.2 4.5
```

erscheint folgende Ausgabe auf dem Bildschirm

```
Parameter 0 = a.out
Parameter 1 = 1.4
Parameter 2 = 3.2
Parameter 3 = 4.5

Die Summe der Kommandozeilen-Parameter : 9.100000
```

Die Angaben in der Kommandozeile sind an das Programm übergeben worden und konnten hier auch verarbeitet werde.

Zeile 3: `main` erhält vom Betriebssystem zwei Parameter. Die Variable `argc` enthält die Anzahl der übergebenen Parameter und `argv[]` die Parameter selbst. Die Namen der Variablen `argc` und `argv` sind natürlich frei wählbar, es hat sich jedoch eingebürgert, die hier verwendeten Bezeichnungen zu benutzen. Sie leiten sich von *arg*ument *c*ount und *arg*ument *v*alues ab.

Bei `argc` ist eine Besonderheit zu beachten. Hat diese Variable zum Beispiel den Wert 1 ist, so bedeutet das, dass kein Kommandozeilen–Parameter eingegeben wurde. Das Betriebssystem übergibt als ersten Parameter nämlich grundsätzlich den Namen des Programms selbst. Also erst wenn `argc` größer als 1 ist, wurde wirklich ein Parameter eingegeben.

Die Deklaration `char *argv[]` bedeutet: Zeiger auf Zeiger auf Zeichen. (Man hätte auch `char **argv` schreiben können). Mit anderen Worten: `argv` ist ein Zeiger, der auf ein Feld zeigt, das wiederum Zeiger enthält. Diese Pointer zeigen schließlich auf die einzelnen Kommandozeilen–Parameter. Die leeren eckigen Klammern weisen darauf hin, dass es sich um ein Feld unbestimmter Größe handelt. Die einzelnen Argumente können durch Indizierung von `argv` angesprochen werden. `argv[1]` zeigt also auf das erste Argument (”1.4”), `argv[2]` auf ”3.2” usw.

Zeile 15: Da es sich bei `argv[i]` um Strings handelt müssen die Eingabeparameter eventuell (je nach ihrer Bestimmung) in einen anderen Typ umgewandelt werden. Dies geschieht in diesem Beispiel mit Hilfe des `sscanf`–Befehls. □

9.7 Wie werden Deklarationen gelesen?

Eine Deklaration besteht grundsätzlich aus einem *Bezeichner* (Variablennamen oder Funktionsnamen), der durch einen oder mehrere *Zeiger*–, *Feld*– oder *Funktions*–Modifikatoren beschrieben wird. Wenn mehrere solcher Modifikatoren miteinander kombinieren, muss man darauf achten, dass Funktionen keine Funktionen oder Felder zurückgeben können und dass Felder auch keine Funktionen als Elemente haben können. Ansonsten sind alle Kombinationen erlaubt. Dabei haben Funktions– und Array–Modifikatoren Vorrang vor Zeiger–Modifikatoren. Durch Klammerung kann diese Rangfolge geändert werden.

Bei der Interpretation beginnt man am besten beim *Bezeichner* und liest nach rechts bis zum Ende usw. bis zu einer einzelnen rechten Klammer. Dann fährt man links vom Bezeichner mit evtl. vorhandenen Zeiger–Modifikatoren fort, bis das Ende oder eine einzelne linke Klammer erreicht wird. Dieses Verfahren wird für jede geschachtelte Klammer von innen nach außen wiederholt. Zum Schluss wird der Typ–Kennzeichner gelesen.

Beispiel 9.8

$$\underbrace{\texttt{char}}_{7}\ \underbrace{*}_{6}\ \underbrace{(*}_{4}\ \underbrace{(*}_{2}\ \underbrace{\texttt{Bezeichner}}_{1}\underbrace{)}\ \underbrace{(\,)}_{3}\ )\ \underbrace{[20]}_{5}$$

Bezeichner (1) ist hier ein Zeiger (2) auf eine Funktion (3) ohne Eingabe–Argumente, die einen Zeiger (4) auf ein Feld mit 20 Elementen (5) zurückgibt, die Zeiger (6) auf `char`–Werte (7) sind! □

Beispiel 9.9 Die folgenden vier Beispiele verdeutlichen den Einsatz von Klammern:

(i)	`char * a[10]`	`a` ist ein Feld der Größe 10 mit Zeigern auf `char`–Werte
(ii)	`char (* a)[10]`	`a` ist Zeiger auf ein Feld der Größe 10 mit `char`–Werte
(iii)	`char *a(int)`	`a` ist Funktion die als Eingabeparameter einen `int`–Wert verlangt und einen Zeiger auf `char`–Wert zurückgibt
(iv)	`char (*a)(int)`	`a` ist Zeiger, auf eine Funktion die als Eingabeparameter einen `int`–Wert verlangt und einen `char`–Wert zurückgibt

□

9.8 Zeiger auf Funktionen

Manchmal ist es nützlich, Funktion an Funktionen zu übergeben. Dies kann mit Hilfe von Zeigern auf Funktionen realisiert werden.

Beispiel 9.10 Zeiger auf Funktionen. In diesem Beispiel wird der Funktion `trapez_regel` ein Zeiger auf die zu integrierende Funktion mitgeliefert. Dadurch ist es möglich, beliebige Funktionen mit Hilfe der Trapez–Regel numerisch zu integrieren. Die Intervallgrenzen und Anzahl der Stützstellen sollen dem Programm durch Kommandozeilen–Parameter übergeben werden.

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

/*************************************************************/
/* Deklaration der Funktion trapez_regel()             */
/* Eingabeparameter : 1.) Zeiger auf Funktion mit
 *                       Eingabeparameter double-Wert
 *                       und double-Rueckgabewert
 *                  2.) double fuer linke Intervallgrenze
 *                  3.) double fuer rechte Intervallgrenze
 *                  4.) int fuer Anzahl der Stuetzstellen
 * Rueckgabewert : double fuer das Integral
*************************************************************/
double trapez_regel(double (*f)(double ),double ,double ,int );

/*************************************************************************/
/* Hauptprogramm                                                         */
/*************************************************************************/
int main(int argc,char** argv)
{
       int n;
       double a,b,integral;

       /* Zeiger auf eine Funktion die als Rueckgabewert
        * eine double-Variable besitzt und als
        * Eingabe eine double-Variable verlangt    */
       double (*fptr)(double);

       if (argc<4)
       {
              printf("Programm ben\"otigt 3 Kommandozeilenparameter :\n");
              printf("1.) Linker Intervallrand (double)\n");
              printf("2.) Rechter Intervallrand (double)\n");
              printf("3.) Anzahl der Teilintervalle fuer");
              printf(" numerische Integration (int)\n");

              return 1;
       }
       else
       {
              sscanf(argv[1],"%lf",&a);
              sscanf(argv[2],"%lf",&b);
              sscanf(argv[3],"%i",&n);


              fptr=(double(*)(double)) cos;/* Zeiger fptr auf cos-Funktion */
              integral=trapez_regel(fptr,a,b,n);
              printf("Das Integral der cos-Funktion ueber das Intervall");
              printf(" [%f , %f]\n",a,b);
              printf("betr\"agt : \t %f (numerisch mit %i",integral,n+1);
              printf(" Stuetzstellen)\n");
              printf("        \t %f (exakt)\n\n",sin(b)-sin(a));

              fptr=(double (*)(double)) sin;/*Zeiger fptr auf sin-Funktion */
              integral=trapez_regel(fptr,a,b,n);
              printf("Das Integral der sin-Funktion ueber das Intervall");
              printf(" [%f , %f]\n",a,b);
              printf("betr\"agt : \t %f (numerisch mit %i",integral,n+1);
```

```
                printf(" Stuetzstellen)\n");
                printf("        \t %f (exakt)\n\n",cos(a)-cos(b));

                return 0;
        }
}

/*********************************************************************/
/* Definition der Funktion trapez_regel()                    */
/*********************************************************************/
double trapez_regel(double (*f)(double ),double a,double b,int n)
{
        n=n+1;
        int k;
        double h=(b-a)/n;
        double integral=0;

        for (k=0;k<=n-1;k++) integral=integral+h/2*(f(a+k*h)+f(a+(k+1)*h));

        return integral;
}
```

□