Elemente der Programmiersprache C

Diese kurzes Kompendium ist keine Programmieranleitung sondern soll in sehr reduzierter Form diejenigen Elemente von C darstellen, die zum Verständnis der Vorlesung bzw. für die Übungsaufgaben unbedingt erforderlich sind. Für weiterführende Studien sei u.a. die Einführung in C (Frau Dr. Meiler) empfohlen.

Konstanten und Variable in C

Wenn wir einen beliebigen mathematischen Ausdruck, Beispiel

$$2(a^2 + 3.21b)$$

für beliebige Werte der Variablen a und b berechnen wollen, dann haben wir damit kaum, der Rechner (Compiler) aber prinzipielle Schwierigkeiten. Der Rechner muss nämlich wissen, mit welchem Typ von Zahlen die Variablen belegt werden sollen und außerdem kann er den Ausdruck in dieser Form nicht verstehen. Für die Typangabe merken wir uns zunächst nur folgendes:

Konstanten: Eine Ziffernfolge wird automatisch als eine Konstante vom Typ **int** (von engl. *integer*) interpretiert. Eine Dezimalzahl in der Form a.b also z.B. 27.54 wird automatisch als eine Konstante vom Typ **double** (von double precision) interpretiert. Daneben gibt es noch eine Reihe weiterer Typen wie **float** (einfache Genauigkeit einer Gleitkommazahl) die unten weiter ausgeführt werden.

Variable: Der Typ (also beispielsweise **int** oder **double**) muss am Beginn der Funktion in der die Variable verwendet werden soll, deklariert werden. Beispiel:

int a,i,ii,i2j,j; double b,x;

Die Syntax (richtige Schreibweise) ist also immer

Typ name1,name2,...;

d.h. Typangabe gefolgt von einer Liste der Variablen diesen Typs, die von einem Semikolon abgeschlossen wird. Die Einträge einer Liste werden durch Komma getrennt. Jeder Name ist eine Folge (begrenzter Länge) von Zeichen, gebildet aus Buchstaben und Ziffern und dem Zeichen_, die nicht mit einer Ziffer beginnen darf. Beispiele: i, i12, alpha_1.

Ausdrücke

Arithmetische Ausdrücke

Der oben angegebene arithmetische Ausdruck ist in der uns aus der Mathematik bekannten Form angegeben. Er enthält Konstante, Variable und mathematische Operationen wie Multiplikation, Addition und Potenzierung. Die Einfassung in Klammern dient dabei der Festlegung der gewünschten Form der Auswertung des Ausdruckes. In C werden die mathematischen Operationen durch Operationssymbole festgelegt. Unser Beispielausdruck involviert nur sog.

Binäre Operatoren:

Diese wirken in der Form

A op B

wobei A und B die Operanden und op den Operator bezeichnet. Letzterer kann dabei ein Element aus der folgenden Liste sein:

$$+ - * / \%$$

(Das Symbol % steht für die modulo Operation, d..h. i%j bezeichnet den Rest der Ganzzahldivision von i durch j.) Wir schreiben dafür auch

$$\mathrm{op} \in \{+, -, *, /, \%\}$$

d.h. op ist ein Element der Menge der aufgeführten Operatoren. Für die Bildung von Ausdrücken über der Menge dieser Operatoren gilt nun folgende rekursive Definition:

- Jede Konstante und jede Variable (und jeder Funktionsaufruf) ist ein Ausdruck.
- Sind A und B Ausdrücke so auch (A) und A op B mit $\mathrm{op} \in \{+, -, *, /, \%\}$

Mit dieser Definition ist jeweils z.B.

i	3.25	i*j	i%j	2*(a*a+3.21*b)

ein Ausdruck.

Neben der binären gibt es auch

Unäre Operatoren :

Diese haben nur einen Operanden und wirken in der Form (Präfixschreibung)

op A

wobei A ein Ausdruck ist. Beispiele for op sind der Vorzeichenoperator + bzw. -. Letzterer verkehrt den Wert eines Ausdruckes in sein Negatives. Wichtig sind

Inkrementoperator: ++ und Dekrementoperator: --

Diese wirken auf Variable (allgemein modifizierbare L-Ausdrücke) und erhöhen bzw. erniedrigen den Wert der Variablen um 1. Sie werden neben der Präfix- vor allem in der Suffixschreibung verwendet

Aop

Der Unterschied besteht in der Reihenfolge der Auswertung, wenn das Konstrukt in einem Ausdruck erscheint, wird bei der Präfigierung op A zuerst die Wertveränderung (um 1) vorgenommen und dann mit dem veränderten Wert von A weitergerechnet und bei der Suffigierung wird die Wertveränderung erst vorgenommen, wenn die Auswertung des Ausdruckes, in dem A op auftritt, abgeschlossen ist. Beispiel: Sei der Wert von i gleich 3. Dann ist

	Wert von i	Wert von 2*++i	Wert von 2*i++
Vorher	3		
Nacher	4	8	6

wobei vorher bzw. nacher die Situation vor und nach der Auswertung des Ausdruckes bezeichnet. Bemerkung: Die Bindungsstärke (s.u.) von ++ ist größer als die von *.

Vergleichs- und logische Ausdrücke

Binäre Ausdrücke können auch mit den Vergleichsoperatoren

$$op \in \{>, >=, <, <=, ==, !=\}$$

gebildet werden. Ein Ausdruck der Form A op B wertet immer zu 1 bzw. 0 aus, wenn die Bedingung erfüllt bzw. nicht erfüllt ist. Beispielweise ist

Ausdruck	Wertet aus zu
3<4	1
4<3	0
3!=4	1
4==4	1
2*3<2*(1+3)	1

usw.

Logische Operatoren sind

&&	Logisches UND
||	Logisches ODER
!	Negation

Die Bewertung der Operanden erfolgt von rechts nach links. Abbruch der Auswertung sobald Wert des Ausdruckes klar.

Zuweisungsausdrücke

Zuweisungsoperatoren sind binäre Operatoren. Die Grundform eines Zuweisungsausdruckes ist

A = B wobei B eine beliebiger Ausdruck und A eine Variable ist (genau eine modifizierbarer L-Wert). Die Auswertung des Ausdruckes erfolgt von rechts nach links, d.h. es wird zunächst die rechte Seite ausgewertet und dieser Wert der Variablen auf der linken Seite zugewiesen. Zu beachten ist, dass der ganze Ausdruck ebenfalls den Wert von B erhält.

Es gibt über den einfache Zuweisungsoperator = hinaus noch eine Reihe weiterer Zuweisungsoperatoren, die als Abkürzungen komplexerer Zuweisungen genutzt werden. Allgemein schreibt man statt

A = A op B wird einfacher A op= B

Die wichtigsten abgekürzten Schreibweisen sind in der folgenden Tabelle dargestellt.

`A += B`	`entspricht`	`A = A + B`
`A -=B`	`entspricht`	`A = A – B`
`A *=B`	`entspricht`	`A = A * B`
`A /=B`	`entspricht`	`A = A / B`
`A %=B`	`entspricht`	`A = A % B`

Bewertung von Ausdrücken

Bei der Abarbeitung eines Programmes werden den Ausdrücken Werte zugewiesen. Dabei werden zuerst die Variablen mit ihren aktuellen Werten belegt (die sich durch die Bewertung der vorhergehenden Ausdrücke bzw. durch deren Initialisierung ergeben) und dann mit diesen Werten sukzessive der Wert des gesamten Ausdrucks bestimmt indem die durch die Operatoren vorgegebenen Operationen ausgeführt werden. Entscheidend ist dabei, dass die Reihenfolge der auszuführenden Operationen durch die Prioritäten der Operatoren festgelegt ist.

Die folgende Tabelle gibt eine Zusammenstellung der Operatoren, wobei diese nach fallendem Range (Priorität) geordnet erscheinen, d.h. Operatoren mit höherer Priorität erscheinen weiter oben in der Tabelle. Mit L bzw. R wird die Reihenfolge der Zusammenfassung bei der Auswertung eines Ausdruckes, in dem diese Operatoren erscheinen, angegeben.

Operatoren	Reihenfolge der Auswertung
`()  []  ->  .`	L
`*  +  -  !  ~  ++  --  &  (typ)  sizeof`	R (unär)
`*  /  %`	L
`+  -`	L
`<<  >>`	L
`<  <=  >  >=`	L
`==  !=`	L
`&`	L
`^`	L
`¦`	L
`&&`	L
`¦¦`	L
`?:`	R
`=  +=  -=  *=  /=  %=  &=  ^=  ¦=  <<=  >>=`	R
`,`	L

unär, monadisch: Operator hat einen Operanden.
binär, dyadisch: Operator hat zwei Operanden.
ternär, triadisch: Operator hat drei Operanden.

Die Operatoren `* + - &` werden **unär** als auch **binär** verwendet!
Das Komma wird als Komma-Operator und auch als Trenner/Punktuator verwendet!
Beim Programmieren immer an Rang und Zusammenfassungsrichtung denken!

Anweisungen

Jedes C Programm besteht aus einer (Folge von) Funktion(en), die wiederum durch eine Folge von Anweisungen definiert sind. Diese werden sequentiell abgearbeitet, wobei die Reihenfolge der Abarbeitung durch sog. Kontrollstrukturen (s.u.) wie Schleifen, bedingte Anweisungen usw. geändert werden kann. Beispiele für Anweisungen sind die

Ausdrucksanweisung: Diese ergibt sich durch einen Ausdruck, der durch ein Semikolon angeschlossen wird. Beispiel: a = 3; Nach Abarbeitung des Ausdrucks a = 3 hat die Variable a den Wert 3. Ein Spezialfall ist die

Leere Anweisung: ;

Block von Anweisungen: Eine Folge von Anweisungen in geschweiften Klammern, z.B. {a = 3; a *= i;}

Felder (Arrays)

Zu jedem Typ T (wie z.B. **int** oder **double**) in kann man einen neuen Typ „Feld vom Typ T“ erzeugen. Ein Feld (man spricht auch von Vektor oder Array) ist damit eine Menge von Objekten gleichen Typs, die in aufeinanderfolgenden Bereichen des Speichers abgelegt werden. Jedes Feldelement wird über einen Index angesprochen, der in eckigen Klammern angegeben wird. Felder müssen wie Variable am Anfang einer Funktion definiert werden. Syntax ist

Typ name[*Zahl der Feldelemente*]

Beispiel:

int alpha[100], beta[10];

definiert die Felder mit Namen alpha und beta bestehend aus 100 bzw. 10 Feldelementen. Nun können wir einem Element des Feldes einen Wert zuweisen:

alpha[3] = 7;

usw.

Zu beachten ist unbedingt, dass in C das erste Element eines Feldes immer den Index 0 hat. Die Elemente des Feldes beta sind also beta[0], beta[1], ... , beta[9]. Für die Einhaltung der Feldgrenzen hat man selbst zu sorgen (beliebte Fehlerquelle).

Der Aufruf eines Feldelementes ist ein Ausdruck und kann somit in einem beliebigen anderen Ausdruck erscheinen (Typverträglichkeit beachten).

Felder können auch mehrdimensional definiert werden.

Funktionen in C

Ein C Programm ist bausteinartig aus Funktionen zusammengesetzt. Es besteht dabei mindestens aus einer Funktion, der Funktion main. Diese dient als geeigneter „Rahmen“ in den der eigentliche Programmtext eingebettet wird . Im einfachsten Fall hat dieser Rahmen die folgende Struktur:

```
#include <stdio.h> // das ist eine Direktive für den Präprozessor = Einbindung der Ein- und Ausgaberoutinen

main()

{
//Hier steht der eigentliche Programmtext

}
```

(Bemerkung: Nach // kann ein beliebiger Kommentar eingefügt werden, d.h. bei der Übersetzung und nachfolgenden Abarbeitung des Programmtextes wird alles was bis zum Zeilenumbruch nach // kommt ignoriert.)

Der Programmtext besteht aus einer Folge von Anweisungen (s.u.), die sich wiederum aufgliedern in

- Lokale Deklarationen
- Eigentliche Anweisungsfolge

Jedes C Programm ist eine Folge von Funktionen (s.u.) und das „Hauptprogramm" main ist selbst eine Funktion, die als Eintrittspunkt in das Programm dient. Bei den lokalen Deklarationen können auch eigene Funktionen eingeführt werden.

Generell ist es angezeigt in komplexeren Programmen eigene Funktionen zu definieren. Das erhöht die Lesbarkeit aber auch die Verlässlichkeit eines Programms, denn die Funktionen können extern auf ihre Richtigkeit getestet werden und dienen dann als verlässliche Bausteine des Gesamtprogramms.
Funktionen in C haben wie Variable einen Typ, weil sie nach ihrem Aufruf im allgemeinen einen Wert zurückliefern. Es gibt also Funktionen vom Typ **int** oder auch **double** usw. Da Funktionen aber auch nur eine Wirkung haben können, ist zusätzlich der Typ **void** (leerer Typ) relevant, der Funktionen zugeordnet wird, die keinen Wert zurückliefern. Das ist z. B. bei unserem Programm main (s.o.) der Fall. Funktionen hängen zudem im allgemeinen von Parametern ab (das sind die Eingabegrößen, für die die Funktionen berechnet werden sollen). Diese haben ebenfalls einen bestimmten Typ. Beispiel für eine mathematische Funktion ist sin(x), die für beliebige reelle Argumente (Werte des Parameters) x definiert ist und die im allgemeinen einen reellen Wert (eine Zahl zwischen 0 und 1) „zurückliefert". Ein anderes Beispiel ist die Funktion GGT(m,n), die eine ganze Zahl, den größten gemeinsamen Teiler der ganzen Zahlen m und n zurückliefert. Als C Objekt betrachtete sind die Parameter der Funktion und die Funktion selbst hier also vom Typ **int**.
Es wird empfohlen (und ist bei rekursiven Definitionen unverzichtbar) Funktionen vor ihrer Definition zu deklarieren. Dazu gibt man vor dem Beginn des gesamten Programms (also noch vor main) den sog. Funktionsprototyp in der Form

typ name(*Liste der Typen der Parameter*)

an. Beispiel:

```
int GGT(int,int);
```

Damit ist dem Compiler bekannt gemacht, dass es eine Funktion mit Namen GGT geben wird, die zwei Parameter vom Typ **int** hat und die einen Wert vom Typ **int** zurückliefert.
Später kann dann die Definition der Funktion erfolgen, bei der die Berechnungen, die die Funktion durchführen soll, festgelegt werden. Diese Definition ist

typ name(*Liste der Parameter*)
{
Anweisungen
}

wobei in den Anweisungen am Anfang im allgemeinen wieder lokale Deklarationen (von lokalen Variablen) stehen.
Für die Bewertung von Ausdrücken, in denen Funktionsaufrufe vorkommen gilt: Jeder Funktionsaufruf ist ein Ausdruck. Der Funktionsaufruf erfolgt in der Form

name(*Liste der aktuellen Parameter*)

Die Übergabe des Funktionswertes an das aufrufende Programm geschieht mit der **return** Anweisung. Diese hat die Syntax

return *ausdruck*;

wobei *ausdruck* ein beliebiger Ausdruck passenden Typs sein kann. Wird diese Anweisung erreicht, wird in das aufrufende Programm zurückgekehrt. Die Funktion liefert damit den Wert von *ausdruck* zurück.

Als Beispiel geben wir ein Programm in dem die Fakultätsfunktion für Berechnungen im Hauptprogramm benötigt wird. Die Fakultät wir dabei mit der rekursiven Definition

$$n!=n(n-1)! \qquad \text{mit } 1!=1$$

berechnet.

```
#include <stdio.h>
int fac(int); //Deklaration der Funktion fac, die die Fakultät einer Zahl berechnet
main(){
int i ;
.
.
.
i = fac(322); // Im Ergebnis hat i den Wert 322!
.
.

}
```

Eine praktische Bemerkung: Für die Übungsaufgaben müssen Sie immer nur die Definition der Funktion angeben, d.h. in dem betreffenden Fall ist das nur das Programmfragment

```
int fac(int n) //Definition der Funktion fac.
{
if (n = = 1) return 1 ; else return n * fac(n-1) ;
}//Ende Definition der Funktion fac.

.
.
.
```

Wird fortgesetzt.