Kapitel 6

Variablen, Datentypen und Operationen

6.1 Deklaration, Initialisierung, Definition

Für die Speicherung und Manipulation von Ein- und Ausgabedaten sowie der Hilfsgrößen eines Algorithmus werden bei der Programmierung Variablen eingesetzt. Je nach Art der Daten wählt man einen von der jeweiligen Programmiersprache vorgegebenen geeigneten Datentyp aus. Vor ihrer ersten Verwendung müssen die Variablen durch Angabe ihres Typs und ihres Namens deklariert werden. In C hat die Deklaration die folgende Form.

```
Datentyp Variablenname;
```

Man kann auch mehrere Variablen desselben Typs auf einmal deklarieren, indem man die entsprechenden Variablennamen mit Komma auflistet:

```
Datentyp Variablenname1, Variablenname2, ...,VariablennameN;
```

Bei der Deklaration können einer Variablen auch schon Werte zugewiesen werden, d.h. eine Initialisierung der Variablen ist bereits möglich. Zusammen mit der Deklaration gilt die Variable dann als definiert.

Die Deklaration von Variablen sollte vor der ersten Ausführungsanweisung stattfinden. Dies ist bei den allermeisten Compilern nicht zwingend notwendig, dient aber der Übersicht des Quelltextes.

Variablennamen

Bei der Vergabe von Variablennamen ist folgendes zu beachten:

- Variablennamen dürfen keine Umlaute enthalten. Als einzigstes Sonderzeichen ist der Unterstrich _ (engl. underscore) erlaubt.
- Variablennamen dürfen Zahlen enthalten, aber nicht mit ihnen beginnen.
- Groß- und Kleinschreibung von Buchstaben wird unterschieden.

6.2 Elementare Datentypen

Die Tabelle 6.1 gibt die Übersicht über die wichtigsten Datentypen in C.

Anhang A widmet sich speziell der Zahlendarstellung im Rechner. Insbesondere werden dort die Begriffe Gleitkommazahl und deren Genauigkeit erörtert.

Schlüsselwort	Datentyp	Anzahl Bytes
char	Zeichen	1
int	ganze Zahl	4
float	Gleitkommazahl mit einfacher Genauigkeit	4
double	Gleitkommazahl mit doppelter Genauigkeit	8
void	leerer Datentyp	

Tabelle 6.1: Elementare Datentypen. Die Bytelänge ist von Architektur zu Architektur unterschiedlich (hier: GNU-C-Compiler unter LINUX für die x86-Architektur. Siehe auch sizeof()).

Beispiel 6.1 Deklaration, Initialisierung, Definition.

```
#include <stdio.h>

int main()
{
    int a=4;
    /* Deklaration von a als ganze Zahl */
    /* + Initialisierung von a, d.h. a wird der Wert 4 zugewiesen */

    printf("Die int-Variable a wurde initialisiert mit %i\n" ,a );

    /* Die Formatangabe %i zeigt an, dass eine int-Variable
       ausgegeben wird */
    return 0;
 }
```

□

Der Datentyp char wird intern als ganzzahliger Datentyp behandelt. Er kann daher mit allen Operatoren behandelt werden, die auch für int verwendet werden. Erst durch die Abbildung der Zahlen von 0 bis 255 auf entsprechende Zeichen (ASCII-Tabelle) entsteht die Verknüpfung zu den Zeichen.
Einige dieser Datentypen können durch Voranstellen von weiteren Schlüsselwörtern modifiziert werden. Modifizierer sind:

- signed/unsigned: Gibt für die Typen int und char an, ob sie mit/ohne Vorzeichen behandelt werden (nur int und char).
- short/long: Reduziert/erhöht die Bytelänge des betreffenden Datentyps. Dabei wirkt sich short nur auf int und long nur auf double aus.
- const: Eine so modifizierte Variable kann initialisiert, aber danach nicht mehr mit einem anderen Wert belegt werden. Die Variable ist „schreibgeschützt“.

Bei den zulässigen Kombinationen ist die Reihenfolge

`const - signed/unsigned - long/short` *`Datentyp Variablenname`*

Beispiel 6.2 Deklaration / Definition von Variablen.
Zulässig:

```
int a;
signed char zeichen1;
unsigned short int b; oder äquivalent unsigned short b;
long double eps;
const int c=12;
```

Im letzten Beispiel wurde der Zuweisungsoperator = (s. Abschnitt 6.4) verwendet, um die schreibgeschützte Variable c zu initialisieren. Variablen vom Typ char werden

durch sogenannte Zeichenkonstanten initialisiert. Zeichenkonstanten gibt man an, indem man ein Zeichen in Hochkommata setzt, z.B.

```
char zeichen1='A';
```

Nicht zulässig:

```
unsigned double d;
long char zeichen1;
char 1zeichen; /* unzulässiger Variablenname */
```

□

Die Funktion `sizeof()` liefert die Anzahl der Bytes zurück, die für einen bestimmten Datentyp benötigt werden. Die Funktion sizeof() hat als Rückgabewert den Typ int.

Beispiel 6.3 C-Anweisung : sizeof().

```
/* Beispiel: sizeof() */
# include <stdio.h>

int main()
{
    printf("Eine int-Variable benoetigt %i Bytes\n", sizeof(int));
    return 0;
}
```

□

6.3 Felder und Strings

6.3.1 Felder

Eine Möglichkeit, aus elementaren Datentypen weitere Typen abzuleiten, ist das Feld (Array). Ein Feld besteht aus n Objekten des gleichen Datentyps. Die Deklaration eines Feldes ist von der Form

```
Datentyp Feldname[n];
```

Weitere Merkmale:

- Die Nummerierung der Feldkomponenten beginnt bei 0 und endet mit n-1.
- Die i-te Komponente des Feldes wird mit Feldname[i] angesprochen, $i = 0, \ldots, n-1$.
- Felder können bei der Deklaration initialisiert werden. Dies geschieht unter Verwendung des Zuweisungsoperators und der geschweiften Klammer.

Beispiel 6.4 Felder.

```
#include<stdio.h>

int main()
{
    float a[3]={3.2, 5, 6};
    /* Deklaration und Initialisierung eines (1 x 3) float-Feldes */

    printf("Die 0.-te Komponente von a hat den Wert %f\n" ,a[0] );
    /* Die Formatangabe %f zeigt an, dass eine float bzw.
    double-Variable ausgegeben wird */
    return 0;
 }
```

□

6.3.2 Mehrdimensionale Felder

Es ist möglich, die Einträge eines Feldes mehrfach zu indizieren und so höherdimensionale Objekte zu erzeugen; für d Dimensionen lautet die Deklaration dann:

`Datentyp Feldname[`n_1`][`n_2`]...[`n_d`];`

Beispiel 6.5 Deklaration und Initialisierung einer ganzzahligen 2 x 3-Matrix.

```
#include <stdio.h>

int main()
{
    int a[2][3]={{1, 2, 3}, {4, 5, 6}};
    printf("Die [0,1].-te Komponente von a hat den Wert %i\n",a[0][1]);
    return 0;
}
```

□

6.3.3 Zeichenketten (Strings)

Eine Sonderstellung unter den Feldern nehmen die Zeichenketten (Strings) ein. Es handelt sich dabei um Felder aus Zeichen:

`char Stringname[Länge];`

Eine Besonderheit stellt dar, dass das Stringende durch die Zeichenkonstante '\0' markiert wird. Der String *Hallo* wird also durch

`char text[]={'H','a','l','l','o','\0'};`

initialisiert.

Ein String kann auch durch

`char text[]="Hallo";`

initialisiert werden. Dieser String hat auch die Länge 6, obwohl nur 5 Zeichen zur Initialisierung benutzt wurden. Das Ende eines Strings markiert immer die Zeichenkonstante '\0'.

Beispiel 6.6 Deklaration und Initialisierung eines Strings.

```
#include <stdio.h>

int main()
{
    char text[]="Hallo";
    printf("%s\n" ,text);

    /* Die Formatangabe %s zeigt an, dass ein String ausgegeben wird. */
    return 0;
}
```

□

6.4 Ausdrücke, Operatoren und mathematische Funktionen

Der Zuweisungsoperator

```
operand1 = operand2
```

weist dem linken Operanden den Wert des rechten Operanden zu.

Beispiel 6.7 Zuweisungsoperator. Zum Beispiel ist im Ergebnis der Anweisungsfolge

```
#include <stdio.h>

int main()
{
        int x,y;
        x=2;
        y=x+4;
        printf("x=%i und y=%i\n",x,y);
        /* Formatangabe %i gibt dem printf-Befehl an,
         * dass an dieser Stelle eine Integervariable
         * ausgeben werden soll. */

        return 0;
}
```

der Wert von x gleich 2 und der Wert von y gleich 6. Hierbei sind x, y, 0, x+4 Operanden, wobei letzterer gleichzeitig ein Ausdruck, bestehend aus den Operanden x, 4 und dem Operator + ist. Sowohl x=2 als auch y=x+4 sind Ausdrücke. Erst das abschließende Semikolon ; wandelt diese Ausdrücke in auszuführende Anweisungen. □

Es können auch Mehrfachzuweisungen auftreten.

Beispiel 6.8 Mehrfachzuweisung. Die folgenden drei Zuweisungen sind äquivalent.

```
#include <stdio.h>

int main()
{
        int a,b,c;

        /* 1. Moeglichkeit */
        a = b = c = 123;
        /* 2. Moeglichkeit */
        a = (b = (c = 123));
        /* 3. Moeglichkeit (Standard) */
        c = 123;
        b=c;
        a=b;

        printf("a=%i, b=%i, c=%i\n",a,b,c);
        return 0;
}
```

□

6.4.1 Arithmetische Operatoren

Unäre Operatoren. Bei unären Operatoren tritt nur ein Operand auf.

Operator	Beschreibung	Beispiel
-	Negation	-a

Binäre Operatoren. Bei binären Operatoren treten zwei Operanden auf. Der Ergebnistyp der Operation hängt vom Operator ab.

Operator	Beschreibung	Beispiel
+	Addition	a+b
-	Subtraktion	a-b
*	Multiplikation	a*b
/	Division (Achtung bei Integerwerten !!!)	a/b
%	Rest bei ganzzahliger Division (Modulooperation)	a%b

Achtung!!! Die Division von Integerzahlen berechnet den ganzzahligen Anteil der Division, z.B. liefert 8/3 das Ergebnis 2. Wird jedoch einer der beiden Operanden in eine Gleitkommazahl umgewandelt, so erhält man das numerisch exakte Ergebnis. z.B. 8.0/3 liefert 2.66666 als Ergebnis (siehe auch Kapitel 6.8).

Analog zur Mathematik gilt " Punktrechnung geht vor Strichrechnung ". Desweiteren werden Ausdrücke in runden Klammern zuerst berechnet.

Beispiel 6.9 Arithmetische Operatoren.

```
   % ermoeglicht das Setzen von mathematischen Ausdruecken
   % wird hier fuer die Referenz benutzt
#include <stdio.h>

int main()
{
       int a,b,c;
       double x;
       a=1;           /* a=1 */
       a=9/8;         /* a=1, Integerdivision */
       a=3.12;        /* a=3, abrunden wegen int-Variable */
       a=-3.12;       /* a=-3 oder -4, Compiler abhaengig */

       b=6;           /* b=6 */
       c=10;          /* c=10 */
       x=b/c;         /* x=0 */
       x=(double) b/c; /* x=0.6 siehe Kapitel 6.8 */
       x=(1+1)/2;     /* x=1 */
       x=0.5+1.0/2;   /* x=1 */
       x=0.5+1/2;     /* x=0.5 */
       x=4.2e12;      /* x=4.2*10^{12} wissenschaftl. Notation */

       return 0;
}
```

□

6.4.2 Vergleichsoperatoren

Vergleichsoperatoren sind binäre Operatoren. Der Ergebniswert ist immer ein Integerwert. Sie liefern den Wert 0, falls die Aussage falsch, und den Wert 1, falls die Aussage richtig ist.

Operator	Beschreibung	Beispiel
>	größer	a>b
>=	größer oder gleich	a>=b
<	kleiner	a<b/3
<=	kleiner oder gleich	a*b<=c
==	gleich (Achtung bei Gleitkommazahlen !!!)	a==b
!=	ungleich (Achtung bei Gleitkommazahlen !!!)	a!=3.14

Achtung !!! Ein typischer Fehler tritt beim Test auf Gleichheit auf, indem statt des Vergleichsoperators == der Zuweisungsoperator = geschrieben wird. Das Prüfen von Gleitkommazahlen auf (Un-)gleichheit kann nur bis auf den Bereich der Maschinengenauigkeit erfolgen und sollte daher vermieden werden.

Beispiel 6.10 Vergleichsoperatoren.

```
#include <stdio.h>

int main()
{
        int a,b;
        int aussage;
        float x,y;

        a=3;                    /* a=3 */
        b=2;                    /* b=2 */
        aussage = a>b;          /* aussage=1 ; entspricht wahr */
        aussage = a==b;         /* aussage=0 ; entspricht falsch */

        x=1.0+1.0e-8;           /* x=1 + 1.0 *10^{-8} */
        y=1.0+2.0e-8;           /* y=1 + 2.0 *10^{-8} */
        aussage = (x==y);       /* aussage=0 oder 1 ; entspricht wahr,
                                   falls eps > 10^{-8}, obwohl x ungleich y */

        return 0;
}
```

□

6.4.3 Logische Operatoren

Es gibt nur einen unären logischen Operator

Operator	Beschreibung	Beispiel
!	logische Negation	! (3>4) /* Ergebnis= 1; entspricht wahr */

und zwei binäre logische Operatoren

Op.	Beschreibung	Beispiel
&&	logisches UND	(3>4) && (3<=4) /* Ergebnis = 0; entspricht falsch */
||	logisches ODER	(3>4) || (3<=4) /* Ergebnis = 1; entspricht wahr */

Die Wahrheitstafeln für das logische UND und das logische ODER sind aus der Algebra bekannt.

6.4.4 Bitorientierte Operatoren (∗)

Bitorientierte Operatoren sind nur auf int-Variablen (bzw. char-Variablen) anwendbar. Um die Funktionsweise zu verstehen, muss man zunächst die Darstellung von Ganzzahlen innerhalb des Rechners verstehen.

Ein Bit ist die kleinste Informationseinheit mit genau zwei möglichen Zuständen:

$$\left\{\begin{array}{l}\text{bit ungesetzt}\\ \text{bit gesetzt}\end{array}\right. \equiv \left\{\begin{array}{l}0\\1\end{array}\right. \equiv \left\{\begin{array}{l}\text{falsch}\\ \text{wahr}\end{array}\right.$$

Ein Byte besteht aus 8 Bit. Eine short int-Variable besteht aus 2 Byte. Damit kann also eine short int-Variable 2^{16} Werte annehme. Das erste Bit bestimmt das Vorzeichen der Zahl. Gesetzt bedeutet - (negativ); nicht gesetzt entspricht + (positiv).

Beispiel 6.11 (Short)-Integerdarstellung im Rechner.

Darstellung im Rechner (binär)	Dezimal
$\underbrace{\underbrace{0}_{+}0000000}_{\text{1. Byte}}\ \underbrace{00001010}_{\text{2. Byte}}$	$2^3 + 2^1 = 10$
$\underbrace{\underbrace{1}_{-}1111111}_{\text{1. Byte}}\ \underbrace{11011011}_{\text{2. Byte}}$	$-(2^5 + 2^2) - 1 = -37$

□

Unäre bitorientierte Operatoren

Operator	Beschreibung	Beispiel
∼	Binärkomplement	∼ a

Binäre bitorientierte Operatoren

Operator	Beschreibung	Beispiel
&	bitweises UND	a & 1
|	bitweises ODER	a | 1
^	bitweises exklusives ODER	a ^ 1
<<	Linksshift der Bits von op1 um op2 Stellen	a << 1
>>	Rechtsshift der Bits von op1 um op2 Stellen	a >> 2

Wahrheitstafel

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Beispiel 6.12 Bitorientierte Operatoren.

```
#include <stdio.h>

int main()
{
        short int a,b,c;

        a=5;            /* 00000000 00000101 = 5 */
        b=6;            /* 00000000 00000110 = 6 */

        c= ~ b;         /* Komplement 11111111 11111001 =-(2^2+2^1)-1=-7 */
        c=a & b;        /* 00000000 00000101 = 5            */
                        /* bit-UND &                        */
                        /* 00000000 00000110 = 6            */
                        /* gleich                           */
                        /* 00000000 00000100 = 4            */

        c=a | b;        /* bit-ODER 00000000 00000111 = 7 */

        c=a^b;          /* bit-ODER exklusiv 00000000 00000011 = 3  */

        c=a << 2;       /* 2 x Linksshift 00000000 00010100 = 20   */

        c=a >> 1;       /* 1 x Rechtsshift & 00000000 00000010 = 2 */

        return 0;
}
```

□

6.4.5 Inkrement- und Dekrementoperatoren

Präfixnotation

Notation	Beschreibung
++ operand	operand=operand+1
−− operand	operand=operand-1

Inkrement– und Dekrementoperatoren in Präfixnotation liefern den inkrementierten bzw. dekrementierten Wert als Ergebnis zurück.

Beispiel 6.13 Präfixnotation.

```
#include <stdio.h>

int main()
{
        int i,j;

        i=3;
        ++i; /* i=4 */
        j=++i; /* i=5, j=5 */

        /* oben angegebene Notation ist aequivalent zu */
        i=3;
        i=i+1;
        i=i+1;
```

```
        j=i;

        return 0;
}
```

□

Postfixnotation

Notation	Beschreibung
operand++	operand=operand+1
operand−−	operand=operand-1

Inkrement– und Dekrementoperatoren in Postfixnotation liefern den Wert vor dem Inkrementieren bzw. Dekrementieren zurück.

Beispiel 6.14 Postfixnotation.

```
#include <stdio.h>

int main()
{
        int i,j;

        i=3;
        i++; /* i=4 */
        j=i++; /* j=4 ,i=5 */

        /* oben angegebene Notatation ist aequivalent zu */

        i=3;
        i=i+1;
        j=i;
        i=i+1;

        return 0;
}
```

□

6.4.6 Adressoperator

Der Vollständigkeit halber wird der Adressoperator “ & “ schon in diesem Kapitel eingeführt, obwohl die Bedeutung erst in Kapitel 8 klar wird.

```
& datenobjekt
```

6.4.7 Prioritäten von Operatoren

Es können beliebig viele Aussagen durch Operatoren verknüpft werden. Die Reihenfolge der Ausführung hängt von der Priorität der jeweiligen Operatoren ab. Operatoren mit höherer Priorität werden vor Operatoren niedriger Priorität ausgeführt. Haben Operatoren die gleiche Priorität so werden sie gemäß ihrer sogenannten Assoziativität von links nach rechts oder umgekehrt abgearbeitet.

Prioritäten von Operatoren beginnend mit der Höchsten

Priorität	Operator	Beschreibung	Assoz.
15	()	Funktionsaufruf	→
	[]	Indizierung	→
	− >	Elementzugriff	→
	.	Elementzugriff	→
14	+	Vorzeichen	←
	-	Vorzeichen	←
	!	Negation	←
	~	Bitkomplement	←
	++	Präfix-Inkrement	←
	−−	Präfix-Dekrement	←
	++	Postfix-Inkrement	←
	−−	Postfix-Dekrement	←
	&	Adresse	←
	∗	Zeigerdereferenzierung	←
	(Typ)	Cast	←
	sizeof()	Größe	←
13	∗	Multiplikation	→
	/	Division	→
	%	Modulo	→
12	+	Addition	→
	-	Subtraktion	→
11	<<	Links-Shift	→
	>>	Rechts-Shift	→
10	<	kleiner	→
	<=	kleiner gleich	→
	>	größer	→
	>=	größer gleich	→
9	==	gleich	→
	! =	ungleich	→
8	&	bitweises UND	→
7	^	bitweises exklusivers ODER	→
6	|	bitweises ODER	→
5	&&	logisches UND	→
4	||	logisches ODER	→
3	:?	Bedingung	←
2	=	Zuweisung	←
	∗ =, / =, + =	Zusammengesetzte Zuweisung	←
	− =, & =, ^ =	Zusammengesetzte Zuweisung	←
	| =, <<= >>=	Zusammengesetzte Zuweisung	←
1	,	Komma-Operator	→

Im Zweifelsfall kann die Priorität durch Klammerung erzwungen werden.

Beispiel 6.15 Prioritäten von Operatoren.

```
#include <stdio.h>

int main()
{
        int a=-4, b=-3, c;

        c=a<b<-1;              /* c=0 ; falsch */
        c=a<(b<-1);            /* c=1 ; wahr */
        c=a ==-4 && b == -2;   /* c=0 ; falsch */

        return 0;
}
```

Die erste Anweisung wird von links nach rechts abgearbeitet. Dabei ist zunächst $a < b == 1$ (wahr). Im nächsten Schritt ist aber $1 < -1 == 0$ (falsch). Die Abarbeitung von rechts erzwingt man mit der Klammer (zweite Zeile). In der dritten Zeile ist der rechte Term neben && falsch. □

6.5 Operationen mit vordefinierten Funktionen

6.5.1 Mathematische Funktionen

Im Headerfile `math.h` werden u.a. Deklarationen der in Tabelle 6.2 zusammengefassten mathematischen Funktionen und Konstanten bereitgestellt:

Funktion/Konstante	Beschreibung
sqrt(x)	Wurzel von x
exp(x)	e^x
log(x)	natürlicher Logarithmus von x
pow(x,y)	x^y
fabs(x)	Absolutbetrag von x : $\|x\|$
fmod(x,y)	realzahliger Rest von x/y
ceil(x)	nächste ganze Zahl $\geq x$
floor(x)	nächste ganze Zahl $\leq x$
sin(x), cos(x), tan(x)	trigonometrische Funktionen
asin(x), acos(x), atan(x)	trig. Umkehrfunktionen
M_E	Eulersche Zahl e
M_PI	π

Tabelle 6.2: Mathematische Funktionen

Für die Zulässigkeit der Operation, d.h. den Definitionsbereich der Argumente, ist der Programmierer verantwortlich, siehe Dokumentationen (`man`). Ansonsten werden Programmabbrüche oder unsinnige Ergebnisse produziert.

Beispiel 6.16 Mathematische Funktionen und Konstanten.

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

int main()
```

```
{
        float x,y,z;
        x=4.5;          /* x=4.5 */
        y=sqrt(x);      /* y=2.121320, was ungefaehr = sqrt(4.5) */
        z=M_PI;         /* z=3.141593, was ungefaehr = pi */

        return 0;
}
```

□

6.5.2 Funktionen für Zeichenketten (Strings)

Im Headerfile `string.h` werden u.a. die Deklarationen der folgenden Funktionen für Strings bereitgestellt:

Funktion	Beschreibung
strcat(s1,s2)	Anhängen von s2 an s1
strcmp(s1,s2)	Lexikographischer Vergleich der Strings s1 und s2
strcpy(s1,s2)	Kopiert s2 auf s1
strlen(s)	Anzahl der Zeichen in String s (= sizeof(s)-1)
strchr(s,c)	Sucht Zeichenkonstante (Character) c in String s

Tabelle 6.3: Funktionen für Strings

Beispiel 6.17 Funktionen für Zeichenketten (Strings).

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

int main()
{
        int i;
        char s1[]="Hallo";    /* reserviert 5+1 Byte im Speicher fuer s1
                              und belegt sie mit H,a,l,l,o,\0     */
        char s2[]="Welt";     /* reserviert 4+1 Byte im Speicher f\"ur s2 */
        char s3[100]="Hallo"; /* reserviert 100 Byte im Speicher f\"ur s3
                               * und belegt die ersten 6 mit H,a,l,l,o,\0 */

        /* !!!NICHT ZULAESSIG!!! (Kann zu Programmabsturz fuehren) *** */
        strcat(s1,s2);        /* Im reservierten Speicherbereich von s1
                              * steht nun H,a,l,l,o,W
                              * Der Rest von s2 wird irgendwo in den
                              * Speicher geschrieben */
        /* ZULAESSIG */
        strcat(s3,s2);        /* Die ersten 10 Bytes von s3 sind nun
                               * belegt mit H,a,l,l,o,W,e,l,t,\0
                               * Der Rest ist zufaellig beschrieben */
        strcpy(s1,s2);        /* Die ersten 5 Bytes von s1 sind nun
                               * belegt mit W,e,l,t,\0 */
        i=strlen(s2);         /* i=4 */
        i=strcmp(s2,s3);      /* i=15, Unterschied zwischen 'W' und 'H' in
                               * ASCII*/
        return 0;
```

```
}
```

□

Achtung! Der Umgang mit Strings ist problematisch, z.B. wird bei dem Befehl strcat(s1,s2) der String s2 an s1 angehängt. Dadurch wird der Speicherbedarf für String s1 vergrößert. Wurde bei der Deklaration von s1 zu wenig Speicherplatz reserviert (allokiert) schreibt der Computer die überschüssigen Zeichen in einen nicht vorher bestimmten Speicherbereich. Dies kann unter Umständen sogar zum Absturz des Programms führen – das Ergebnis können seltsame und schwer zu findende Fehler im Programm sein, die teilweise nicht immer auftreten (siehe auch Beipiel 6.17).

6.6 Zusammengesetzte Anweisungen

Wertzuweisungen der Form

```
op1=op1 operator op2;
```

können zu

```
op1 operator = op2;
```

verkürzt werden.
Hierbei ist *operator* $\in \{+, -, *, /, \%, |, \hat{}, <<, >>\}$.

Beispiel 6.18 Zusammengesetzte Anweisungen.

```
#include <stdio.h>

int main()
{
        int i=7,j=3;

        i += j; /* i=i+j; */
        i >>= 1; /* i=i >> 1 (i=i/2), bitorientierte Operation */
        j *= i; /* j=j*i */

        return 0;
}
```

□

6.7 Nützliche Konstanten

Für systemabhängige Zahlenbereiche, Genauigkeiten usw. ist die Auswahl der Konstanten aus Tabelle 6.4 und Tabelle 6.5 recht hilfreich. Sie stehen dem Programmierer durch Einbinden der Headerdateien `float.h` bzw. `limits.h` zur Verfügung.

Weitere Konstanten können in der Datei `float.h` nachgeschaut werden. Der genaue Speicherort dieser Datei ist abhängig von der gerade verwendeten Version des gcc und der verwendeten Distribution. Die entsprechenden Headerfiles können auch mit dem Befehl

```
find /usr -name float.h -print
```

gesucht werden. Dieser Befehl durchsucht den entsprechenden Teil des Verzeichnisbaums (`/usr`) nach der Datei namens `float.h`.

Tabelle 6.4: Konstanten aus float.h

Konstante	Beschreibung
FLT_DIG	Anzahl gültiger Dezimalstellen für float
FLT_MIN	Kleinste, darstellbare positive float Zahl
FLT_MAX	Größte, darstellbare positive float Zahl
FLT_EPSILON	Kleinste positive Zahl mit $1.0 + eps \neq 1.0$
DBL_	wie oben für double
LDBL_	wie oben für long double

Tabelle 6.5: Konstanten aus limits.h

Konstante	Beschreibung
INT_MIN	Kleinste, darstellbare int Zahl
INT_MAX	Größte, darstellbare int Zahl
SHRT_	wie oben für short int

6.8 Typkonversion (cast)

Beispiel 6.19 Abgeschnittene Division.

```
#include <stdio.h>

int main()
{
        int a=10, b=3;
        float quotient;
        quotient = a/b;         /* quotient = 3 */
        quotient = (float) a/b; /* quotient = 3.3333 */

        return 0;
}
```

Nach der Zuweisung `a/b` hat die Variable `quotient` den Wert 3.0, obwohl sie als Gleitkommazahl deklariert wurde! Ursache: Resultat der Division zweier int-Variablen ist standardmäsig wieder ein int-Datenobjekt.

Abhilfe schaffen hier Typumwandlungen (engl.: Casts). Dazu setzt man den gewünschten Datentyp in Klammern vor das umzuwandelnde Objekt, im obigen Beispiel:

```
quotient = (float) a/b;
```

Hierdurch wird das Ergebnis mit den Nachkommastellen übergeben. □

Achtung! bei Klammerung von Ausdrücken! Die Anweisung

```
quotient = (float) (a/b);
```

führt wegen der Klammern die Division komplett im int-Kontext durch und der Cast bleibt wirkungslos.

Bemerkung 6.20 Die im ersten Beispiel gezeigte abgeschnittene Division erlaubt in Verbindung mit dem Modulooperator % eine einfache Programmierung der Division mit Rest. *Übungsaufgabe* □

Ist einer der Operanden eine Konstante, so kann man auch auf Casts verzichten: Statt

```
quotient = (float) 10/b;
```

kann man die Anweisung

```
quotient = 10.0/b;
```

verwenden.

6.9 Standardein– und –ausgabe

Eingabe: Das Programm fordert benötigte Informationen/Daten vom Benutzer an.

Ausgabe: Das Programm teilt die Forderung nach Eingabedaten dem Benutzer mit und gibt (Zwischen–) Ergebnisse aus.

6.9.1 Ausgabe

Die Ausgabe auf das Standardausgabegerät (Terminal, Bildschirm) erfolgt mit der `printf()`-Bibliotheksfunktion. Die Anweisung ist von der Form

```
printf(Formatstringkonstante, Argumentliste);
```

Die Argumentliste ist eine Liste von auszugebenden Objekten, jeweils durch ein Komma getrennt (Variablennamen, arithmetische Ausdrücke etc.). Die Formatstringkonstante enthält neben Text zusätzliche spezielle Zeichen: spezielle Zeichenkonstanten (Escapesequenzen) und Formatangaben.

Zeichenkonstante	**erzeugt**
\n	neue Zeile
\t	Tabulator
\v	vertikaler Tabulator
\b	Backspace
\\	Backslash \
\?	Fragezeichen ?
\’	Hochkomma
\”	Anführungsstriche

Die Formatangaben spezifizieren, welcher Datentyp auszugeben ist und wie er auszugeben ist. Sie beginnen mit `%`. Die folgende Tabelle gibt einen Überblick über die wichtigsten Formatangaben:

Formatangabe	Datentyp
%f	float, double
%i, %d	int, short
%u	unsigned int
%o	int, short oktal
%x	int, short hexadezimal
%c	char
%s	Zeichenkette (String)
%li, %ld	long
%Lf	long double
%e	float, double wissenschaftl. Notation

Durch Einfügen eines Leerzeichens nach % wird Platz für das Vorzeichen ausgespart. Nur negative Vorzeichen werden angezeigt. Fügt man stattdessen ein + ein, so wird das Vorzeichen immer angezeigt.

Weitere Optionen kann man aus Beispiel 6.21 entnehmen.

Beispiel 6.21 Ausgabe von Gleitkommazahlen.

```
#include <stdio.h>

int main()
{
       const double pi=3.14159265;

       printf("Pi = %f\n",pi);
       printf("Pi = % f\n",pi);
       printf("Pi = %+f\n",pi);
       printf("Pi = %.3f\n",pi);
       printf("Pi = %.7e\n",pi);

       return 0;
}
```

erzeugen die Bildschirmausgabe

Pi = 3.141593
Pi = 3.141593
Pi = +3.141593
Pi = 3.142
Pi = 3.1415927e+00

□

6.9.2 Eingabe

Für das Einlesen von Tastatureingaben des Benutzers steht u.a. die Bibliotheksfunktion `scanf()` zur Verfügung. Ihre Verwendung ist auf den ersten Blick identisch mit der von `printf()`.

```
scanf(Formatstringkonstante, Argumentliste);
```

Die Argumentliste bezieht sich auf die Variablen, in denen die eingegebenen Werte abgelegt werden sollen, wobei zu beachten ist, dass in der Argumentliste nicht die Variablen selbst, sondern ihre *Adressen* anzugeben sind. Dazu verwendet man den Adressoperator `&`.

Beispiel 6.22 Einlesen einer ganzen Zahl.

```
#include <stdio.h>

int main()
{
       int a;

       printf("Geben Sie eine ganze Zahl ein: ");
       scanf("%i",&a);
       printf("a hat nun den Wert : %i\n",a);
```

```
        return 0;
}
```

Die eingegebene Zahl wird als int interpretiert und an der Adresse der Variablen a abgelegt. □

Die anderen Formatangaben sind im Wesentlichen analog zu `printf()`. Eine Ausnahme ist das Einlesen von double- und long double-Variablen. Statt %f sollte man hier

- %lf für double
- %Lf für long double

verwenden. Das Verhalten variiert je nach verwendetem C-Compiler.

Achtung! Handelt es sich bei der einzulesenden Variable um ein Feld (insbesondere String) oder eine Zeiger Variable (siehe Kapitel 8), so entfällt der Adressoperator & im scanf()-Befehl.

Ein Beispiel:

```
char text[100];
scanf("\%s",text);
```

Die Funktion `scanf` ist immer wieder eine Quelle für Fehler.

```
int zahl;
char buchstabe;

scanf("%i", &zahl);
scanf("%c", &buchstabe);
```

Wenn man einen solchen Code laufen lässt, wird man sehen, dass das Programm den zweiten `scanf`-Befehl scheinbar einfach überspringt. Der Grund ist die Art, wie `scanf` arbeitet. Die Eingabe des Benutzers beim ersten `scanf` besteht aus zwei Teilen: einer Zahl (sagen wir 23) und der Eingabetaste (die wir mit '\n' bezeichnen). Die Zahl 23 wird in die Variable `zahl` kopiert, das '\n' steht aber immer noch im sog. Tastaturpuffer. Beim zweiten `scanf` liest der Rechner dann sofort das '\n' aus und geht davon aus, dass der Benutzer dieses '\n' als Wert für die Variable `buchstabe` wollte. Vermeiden kann man dies mit einem auf den ersten Blick komplizierten Konstrukt, das dafür deutlich flexibler ist.

```
int zahl;
char buchstabe;
char tempstring[80];

/* wir lesen eine ganze Zeile in den String tempstring von stdin --
 * das ist die Standardeingabe
 */
fgets(tempstring, sizeof(tempstring), stdin);

/* Wir haben jetzt einen ganzen String, wie teilen wir ihn auf?
 * => mit der Funktion sscanf
 */
sscanf(tempstring, "%d", &zahl);

/* und nun nochmal fuer den Buchstaben */
```

```
fgets(tempstring, sizeof(tempstring), stdin);
sscanf(tempstring, "%c", &buchstabe);
```

Der Rückgabewert von `fgets` ist ein Zeiger; der obige Code überprüft nicht, ob dies ein NULL Zeiger ist – diese Überprüfung ist in einem Programm natürlich Pflicht! Die Funktionen `fgets` und `sscanf` sind in `stdio.h` deklariert.