Programmieren in C

P. Bohl
Email : peter.bohl@gmail.com

Programmieren in C

1. Grundlagen

1.1 Programmstruktur

Das erste C-Programm

BEISPIELE/b11a.c:

```
#include <stdio.h>

main()          /* Ausgabe: Mein erstes C-Programm! */
{
        printf("Mein erstes C-Programm!\n");
}
```

Erläuterungen:

#include	= Einfügen von Quelldateien
<stdio.h>	= Headerfile
main	= Funktion
()	= Funktionsoperatoren
printf	= Funktionsaufruf
{}	= Funktionskörper(freies Format)
/* Ausgabe: ... */	= Kommentar (Kommentare nicht schachteln)

1.1 Programmstruktur (Programm)

Ein Programm in C besteht aus einer Menge von Funktionen, die in einem oder in mehreren Quelltextfiles untergebracht sein können. Funktionen werden nicht geschachtelt. Außerdem muss jedes C-Programm eine main-Funktion haben. Bei dieser Funktion beginnt die Programmabarbeitung. Nach dem Funktionsnamen folgt, in runde Klammern eingeschlossen, die Liste der formalen Parameter. Sie werden voneinander durch Komma getrennt. Alle zu einer Funktion gehörenden Anweisungen müssen in geschweifte Klammern eingeschlossen werden. Jede einzelne Anweisung wird durch ein Semikolon beendet.

1.1 Programmstruktur (Programm)

BEISPIELE/b11b.c:

```
#include <stdio.h>

main()              /* Ausgabe: Mein 2. C-Programm! */
{
     ausgabe(2);
}

ausgabe(wert)
int wert;
{
     printf("Mein %d. ",wert);
     printf("C-Programm!\n");
}
```

notwendige Erläuterungen zum Beispiel:

printf	= Funktion für formatierte Ausgabe
%d	= Ausgabeformat für Integerwerte
%c	= Ausgabeformat für Zeichen
%lf	= Ausgabeformat für Gleitkommazahlen
\n	= Zeilenvorschub
ausgabe(2)	= Aufruf Funktion ausgabe mit Argument 2
ausgabe(wert)	= Vereinbarung Funktion ausgabe mit Parameter wert

1.1 Programmstruktur (Trennzeichen)

C-Programme können formatfrei geschrieben werden, d.h. eine bestimmte Zeilenstruktur ist nicht erforderlich.Außerhalb von Zeichen- oder Zeichenkettenkonstanten werden

- Leerzeichen
- Tabulator
- neue Zeile (nl)
- Kommentar

als Trennzeichen zwischen den einzelnen Grundelementen der Sprache betrachtet.

1.2.1 Variable

- eine Variable ist symbolische Repräsentation von Speicherplatz
- Typ und Speicherklasse werden vergeben
- die Vereinbarung einer Variable ist vor ihrer 1. Benutzung notwendig

Beispiel

```
int x ;          /* x ist Variable für ganze Zahlen */

float y, z;      /* y,z sind Variablen für reelle Zahlen */
```

Allgemeine Form einer Variablenvereinbarung

```
speicherklasse typ bezeichner1, ..., bezeichnern;
```

1.2.1 Form einer Variablenvereinbarung

- Speicherklasse
  - *auto, register, static, extern.*
    Die "Speicherklasse" einer Variablen bestimmt die Lebensdauer einer Variablen.
- Typ:
  - Der Typ (Datentyp) legt die Größe und Struktur des Speichers fest, welcher über die Variable angesprochen wird.
- Bezeichner:
  - Legt den Name fest, über den der zugehörige Speicher angesprochen (gelesen, beschrieben) wird.

1.2.2 Bezeichner

Bezeichner dienen zur Identifikation von Objekten innerhalb eines C-Programmes legen einen Variablennamen, Funktionsnamen, usw. fest beliebig lange alpha-numerische Zeichenfolge einschließlich _ erstes Zeichen muss ein Buchstabe sein, _ gilt als Buchstabe es wird zwischen Klein- und Großbuchstaben unterschieden

- Kleinbuchstaben in Variablenbezeichnern,
- Großbuchstaben in Bezeichnern für symbolische Konstanten
- Bezeichner mit einem _ am Anfang sind für Bibliotheksfunktionen

1.2.2 Regeln für Bezeichner

Für einen gültigen Bezeichner gibt es somit folgende Regeln:

- Namen bestehen aus Buchstaben, Ziffern und Unterstrichen.
- Das erste Zeichen eines Bezeichners muss ein Buchstabe sein.
- Bezeichner sollten nicht mit einem Unterstrich beginnen, da solche Bezeichner gewöhnlich für das System reserviert sind. Dies ist aber wohl eher eine Stil-Frage als eine Regel.
- Es wird zwischen Groß- und Kleinbuchstaben unterschieden.
- Schlüsselwörter von C dürfen nicht als Bezeichner verwendet werden.

1.2.3 Schlüsselwörter

Schlüsselwörter sind Bezeichner mit einer vorgegebenen Bedeutung in C. Sie dürfen nicht anderweitig verwendet werden. So dürfen Sie beispielsweise keine Variable mit dem Bezeichner »int« verwenden, da es auch einen Basisdatentyp hierzu gibt

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

1.2.4 Literale

Als Literale werden Zahlen, Zeichenketten und Wahrheitswerte im Quelltext bezeichnet, die ebenfalls nach einem bestimmten Muster aufgebaut sein müssen. Man kann auch sagen:

Literale sind von einer Programmiersprache definierte Zeichenfolgen zur Darstellung der Werte von Basistypen.

1.2.4 Integer-Konstanten

dezimal	15	-32768	100l	5789L
oktal	017	0100000	0144l	013235L
hexadezimal	0xf	0x8000	0x64l	0X169DL

ANSI-C: const int ci = 0xff /* Integer-Konstante */

ANSI-C: 32768u /* unsigned */

```
#include <stdio.h>

main()
{
     printf("%d %d %d\n",33,033,0x33);
}
```

1.2.4 Real-Konstanten

1.23	.25	3.1415926
.123E+1	2.5e-1	314159.26E-5
123E-2	25e-2	31415926E-7

ANSI-C: 3.7e-9f /* float - Standard ist double */

```
#include <stdio.h>

main()
{
     printf("%lf %lg\n",1.23, 2.5e-1);
}
```

1.2.4 Zeichenkonstanten

```
'0'     /* Wert 0 (ASCII 48) */
'A'     /* Buchstabe A (ASCII 65) */
'\0'    /* Nullzeichen (NUL) */
'\n'    /* neue Zeile (nl) */
'\t'    /* Tabulator */
'\b'    /* Backspace */
'\f'    /* Seitenvorschub */
'\r'    /* Zeilenanfang */
'\v'    /* Vertikal-Tabulator */
'\''    /* Apostroph */
'\\'    /* Backslash */
```

1.2.4 Zeichenkettenkonstanten

- sind eine in Ausführungszeichen (") eingeschlossene Folge von null, einem oder mehreren Zeichen
- besitzt den Typ "Zeichenfeld"
- wird mit dem Zeichen '\0' (NULL) abgeschlossen

==> Zeichenkettenkonstante ist ein Byte länger als die sichtbare Zeichenanzahl !!

- `"\n So ein schoener Tag! \n"`
- `"A"`
- `" "` `/* leere Zeichenkette*/`
- `"Zeichenf1" " Zeichenf2"` `/* => "Zeichenf1 Zeichenf2" */`
- `"Zeichenfolge ueber \`
`zwei Zeilen"`

1.2.4 Symbolische Konstanten

Symbolische Bezeichner für Konstanten verbessern Lesbarkeit und Modifizierbarkeit von Programmen. Sie werden durch einen sogenannten C-Präprozessor ausgewertet.

allg.: #define name string

- #define N 512
- #define MAX 5*N
- #define EOF (-1)
- #define NL '\n'
- #define begin {
- #define end }
- #define print(a) printf(#a) // -> „a“

1.2.5 Kommentare

Kommentare sind Textteile in einem C-Quelltext, die vom Compiler ignoriert werden. Kommentare können an einer beliebigen Stelle im Quelltext stehen. Kommentare können auf eine Programmzeile beschränkt sein oder sich über mehrere Zeilen erstrecken.

```c
#include <stdio.h>

int main (void) {       //Beginn des Hauptprogramms
   int i = 10;          //Variable int mit dem Namen i und Wert 10
   printf("%d",i);      //Gibt die Zahl 10 aus.
   printf("\n");        //Springt eine Zeile weiter.
   printf("10");        //Gibt den String "10" aus.
   return 0;

/*  Hier sehen Sie noch eine 2. Möglichkeit, Kommentare
    einzufügen. Dieser Kommentar wird mit einem Slash-
    Sternchen eröffnet und mit einem Sternchen-Slash
    wieder beendet. Alles dazwischen wird vom Compiler
    ignoriert.  */
}
```

1.3 Ausgabe und Eingabe

Motivation:

In diesem Abschnitt werden einige Funktionen zur Ein- bzw. Ausgabe von Daten vorgestellt, die zum Schreiben einfacher Programme nützlich aber mit den bisher behandelten Sprachmitteln noch nicht vollständig erklärbar sind.

1.3 Ausgabe und Eingabe

Eingabe: c=getchar();

Ausgabe: putchar(c);

```
/* Dieses Programm liest zeichenweise von der Standardeingabe */
/* ( bis zum Erreichen der End-Of-File-Bedingung - EOF )      */
/* und gibt diese Zeichen wieder in die Standardausgabe aus.  */

#include <stdio.h>

main()
{
     int c;

     while(( c = getchar()) != EOF) /* beachte Klammern ! */
          putchar(c);
}
```

1.3.1 printf

printf("format_string", arg1, arg2, ...);

Die mit dem %-Zeichen eingeleiteten Formatelemente greifen nacheinander auf die durch Komma getrennten Parameter zu (das erste %i auf 3, das zweite %i auf 2 und %s auf den String "Fünf").

```
printf("%i plus %i ist gleich %s.\n", 3, 2, "Fünf");
```

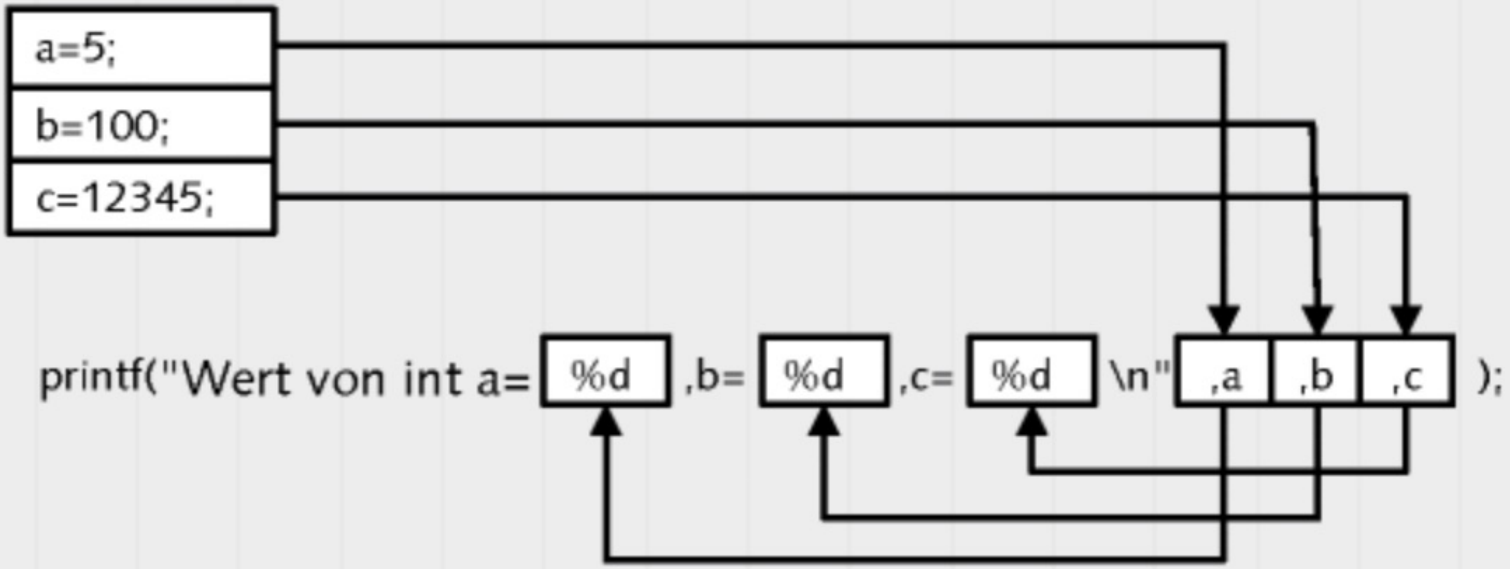
➢ `3 plus 2 ist gleich Fünf.`

Innerhalb von format werden Umwandlungszeichen (engl. conversion modifier) für die weiteren Parameter eingesetzt. Hierbei muss der richtige Typ verwendet werden. Die wichtigsten Umwandlungszeichen sind....

1.3.1 printf

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

int main(void) {
   int a=5;
   int b=100, c=12345;
   printf("Wert von int a=%d ,b=%d, c=%d\n", a, b, c);
   return 0;
}
```



1.3.1 Umwandlungen

Zeichen	Umwandlung
%d oder %i	int
%c	einzelnes Zeichen
%e oder %E	double im Format [-]d.ddd e±dd bzw. [-]d.ddd E±d
%f	float im Format [-]ddd.ddd
%lf	double im Format [-]ddd.ddd
%o	int als Oktalzahl ausgeben
%p	die Adresse eines Zeigers
%s	Zeichenkette ausgeben
%u	unsigned int
%lu	long unsigned
%x oder %X	int als Hexadezimalzahl ausgeben
%%	Prozentzeichen

1.3.1 Flags

Neben dem Umwandlungszeichen kann eine Umwandlungsangabe weitere Elemente zur Formatierung erhalten. Dies sind maximal:

- ein Flag
- die Feldbreite
- durch einen Punkt getrennt die Anzahl der Nachkommstellen (Längenangabe)
- und an letzter Stelle schließlich das Umwandlungszeichen selbst

1.3.1 Flags

Flags:

Unmittelbar nach dem Prozentzeichen werden die Flags (dt. Kennzeichnung) angegeben. Sie haben die folgende Bedeutung:

- - (Minus): Der Text wird links ausgerichtet.
- + (Plus): Es wird auch bei einem positiven Wert ein Vorzeichen ausgegeben.
- Leerzeichen: Ein Leerzeichen wird ausgegeben, wenn der Wert positiv ist. (unsichtbares +)
- # : Welche Wirkung das Kennzeichen # hat, ist abhängig vom verwendeten Format: Wenn ein Wert über %x als Hexadezimal ausgegeben wird, so wird jedem Wert ein 0x vorangestellt (außer der Wert ist 0).
- 0 : Die Auffüllung erfolgt mit Nullen anstelle von Leerzeichen, wenn die Feldbreite verändert wird.

1.3.1 Beispiel Flags

```
int main()
  {
     printf("Zahl 67:%+i\n", 67);
     printf("Zahl 67:% i\n", 67);
     printf("Zahl 67:%#x\n", 67);
     printf("Zahl 0:%0x\n", 0);
     return 0;
  }
```

- Zahl 67:+67
- Zahl 67: 67
- Zahl 67:0x43
- Zahl 0:0

1.3.1 Breite, Nachkommastellen

Feldbreite:

Hinter dem Flag kann die Feldbreite (engl. field width) festgelegt werden. Das bedeutet, dass die Ausgabe mit der entsprechenden Anzahl von Leerzeichen aufgefüllt wird.

Nachkommastellen

Nach der Feldbreite folgt, durch einen Punkt getrennt, die Genauigkeit. Bei %f werden ansonsten standardmäßig 6! Nachkommastellen ausgegeben. Diese Angaben sind natürlich auch nur bei den Gleitkommatypen float und double sinnvoll, weil alle anderen Typen keine Nachkommastellen besitzen.

1.3.1 Beispiel Breite

Beispiel: Feldreite

```
int main()

{
   printf("Zahlen rechtsbündig ausgeben: %5d, %5d, %5d\n",34,
   343, 3343);
   printf("Zahlen rechtsbündig ausgeben, links mit 0
   aufgefüllt: %05d, %05d, %05d\n",34, 343, 3343);
   printf("Zahlen linksbündig ausgeben: %-5d, %-5d, %-
   5d\n",34, 343, 3343);
   return 0;

}
```

- Zahlen rechtsbündig ausgeben: 34, 343, 3343
- Zahlen rechtsbündig ausgeben, links mit 0 aufgefüllt: 00034, 00343, 03343
- Zahlen linksbündig ausgeben: 34 , 343 , 3343

1.3.1 Beispiel Nachkommastellen

Beispiel: Nachkommastellen

```
int main()

 {

  double betrag1 = 1.5634323;
  double betrag2 = 10.2432422;
  printf("Summe: %7.3f\n", betrag1 + betrag2);
  return 0;

 }
```

- Summe: 011.807

1.3.4 Formatierte Eingabe mit scanf

Mit der Funktion scanf() können Werte unterschiedlicher Datentypen formatiert eingelesen werden. Eingelesen wird dabei von der Standardeingabe (stdin). Mit Standardeingabe ist normalerweise die Tastatur gemeint.

scanf() ist ähnlich aufgebaut wie printf(). Wie bei printf() werden hier zwei Klammern und zwei Hochkommata verwendet. Es wird also formatiert eingelesen. Das Formatzeichen %d steht für die formatierte Eingabe einer dezimalen Zahl.

1.3.2 Formatierte Eingabe mit scanf

scanf("format_string", & arg1, ...);

format_string: ohne Text

Formate ähnlich Ausgabe

```
int n,i;  double d;  char c;
n=scanf("%d %lf %c",&i,&d,&c);
```

Eingabe: 5 3.7e-2 x

Ergebnis: i:=5

d:=0.037

c:='x' /* kein Trennzeichen! */

n:=3 /* Anzahl Konvertierungen */

1.3.2 Formatierte Eingabe mit scanf

Was bedeutet hier das Zeichen »&«?

Eine Variable kann in die vier folgenden Einzelteile zerlegt werden:

- Datentyp
- Name der Variable
- Speicheradresse der Variable
- Wert der Variable

1.3.2 Formatierte Eingabe mit scanf

```
int i;
printf("Bitte geben Sie eine Zahl ein : ");
scanf("%d",&i);   /* Wartet auf die Eingabe.*/
```

Der Datentyp ist int, der Name ist i, und die Adresse wird während der Laufzeit zugewiesen (darauf haben Sie keinen Einfluss). Die Speicheradresse sei hier z. B. 0000:123A. Der Wert ist der, den Sie mit scanf() noch eingeben mussten. Wurde jetzt z. B. 5 eingegeben, ist dieser Speicherplatz wie folgt belegt:

Das &-Zeichen ist nichts anderes als der Adressoperator. Dies bedeutet hier, dass der Variablen i vom Typ int mit der Speicheradresse 0000:123A der Wert 5 zugewiesen wird.

Datentyp	Name	Speicher-adresse	Wert
int	i	0000:123A	5

1.4 Grunddatentypen

Motivation:

Mit dem **Datentyp** werden beliebigen Datenelementen Eigenschaften wie integraler, reeller oder komplexer Typ zugeordnet und Festlegungen für die Speicherung der Daten getroffen. Datentypen werden Variablen, Konstanten, Funktionen usw. zugeordnet.

1.4 Basisdatentypen

Zu den Grundlagen der C-Programmierung gehört auch die Kenntnis der einzelnen Datentypen. Datentypen sind, wie der Name schon vermuten lässt, Arten von Variablen, in denen Sie Daten speichern können, um zu einem späteren Zeitpunkt wieder darauf zurückzugreifen. Diese Variablen bestehen aus zwei Teilen:

- dem Datentyp, der eine bestimmte Menge Arbeitsspeicher zugewiesen bekommt,
- und dem Namen der Variable, mit dem dieser Datentyp im Programm angesprochen werden kann.

```
Typ name;
Typ name1, name2, name3;
```

Als Basisdatentypen werden einfache vordefinierte Datentypen bezeichnet. Dies umfasst in der Regel Zahlen (int, short int, long int, float, double und long double), Zeichen (char, wchar_t) und den (Nichts-)Typ (void).

1.4 Was ist eine Variable?

Eine Variable ist eine Stelle (Adresse) im Hauptspeicher (RAM), an der Sie einen Wert ablegen können und gegebenenfalls später wieder darauf zurückgreifen können. Neben einer Adresse hat eine Variable auch einen Namen, genauer gesagt einen Bezeichner, mit dem man auf diesen Wert namentlich zugreifen kann. Und natürlich belegt eine Variable auch eine gewisse Größe des Hauptspeichers, was man mit dem Typ der Variablen mitteilt. Rein syntaktisch kann man das wie folgt ausdrücken:

```
long lvar;
```

Hier haben Sie eine Variable mit dem Namen (Bezeichner) `lvar` vom Typ `long`, der üblicherweise vier Bytes (auf 32-Bit-Systemen) im Hauptspeicher (RAM) belegt. Wo (d. h. an welcher Speicheradresse) im Arbeitsspeicher Speicherplatz für diese Variable reserviert wird – hier vier Bytes –, können Sie nicht beeinflussen.

1.4 Erlaubte Bezeichner

Mit dem Begriff Bezeichner werden Namen für Variablen, Funktionen, Datentypen und Makros zusammengefasst. Damit Sie bei der Namensvergabe von Variablen oder (später) Funktionen keine Probleme bekommen, müssen Sie bei deren Angabe folgende Regeln beachten:

Ein Bezeichner darf aus einer Folge von Buchstaben, Dezimalziffern und Unterstrichen bestehen. Einige Beispiele:

```
var8, _var, _666, var_fuer_100tmp, VAR, Var
```

C unterscheidet zwischen Groß- und Kleinbuchstaben.

```
Var, VAr, VAR, vAR, vaR, var
```

Hierbei handelt es sich jeweils um verschiedene Bezeichner.

Das erste Zeichen darf keine Dezimalzahl sein.

Die Länge des Bezeichners ist beliebig lang. Nach ANSI-C-Standard sind aber nur die ersten 31 Zeichen von Bedeutung. Allerdings können viele Compiler auch zwischen mehr Zeichen unterscheiden.

1.4 Basisdatentypen

Datentyp	16-Bit-Rechner	32-Bit-Rechner
char	8 Bit	8 Bit
short oder **short int**	16 Bit	16 Bit
int	16 Bit	32 Bit
long oder **long int**	32 Bit	32 Bit
float	32 Bit	32 Bit
double oder **long float**	64 Bit	64 Bit

```
#include <stdio.h>

main()
{
     int i,k;
     unsigned int u;

     k=2147483647;
     i=k + 2;
     u=k + 2;
     printf("i:%x %d\n",i,i);
     printf("u:%x %u\n",u,u);
}
```

es gibt den speziellen Typ: **unsigned** (vorzeichenlos):

- **unsigned char**, **unsigned short**, **unsigned long int**
- in ANSI-C zusätzlich: **signed** (**char**), **volatile**

1.4.1 Integer

Der Datentyp `int` muss, gemäß ANSI C, mindestens eine Größe von zwei Byte aufweisen. Mit diesen zwei Bytes lässt sich ein Zahlenraum von -32768 bis +32767 beschreiben. Mit dem Datentyp `int` lassen sich nur Ganzzahlen darstellen. Die Abkürzung `int` steht für Integer.

Hier kommen Sie auch gleich mit betriebssystemspezifischen Eigenheiten in Berührung. Auf 16-Bit-Systemen mag das eben Gesagte zutreffen. Dort ist ein Integer (`int`) auch wirklich zwei Bytes groß.

Auf Betriebssysteme auf 32-Bit-Basis entspricht der Integer vier Byte. Somit erstreckt sich der Zahlenraum auf 32-Bit-Systemen von -2147483648 bis +2147483647.

Ein `int` hat somit laut Standard die natürliche Größe, die von der »Ausführ-Umgebung« vorgeschlagen wird.

1.4.1 Integer

Sicherlich stellen Sie sich jetzt die Frage, was ist dann mit der neuen 64-Bit-Architektur? Theoretisch hätte hier `int` ja eine Wortbreite von 64 Bit. Auch die Zeiger hängen entscheidend von der Wortbreite ab.

Daher hat man beim Übergang von der 32-Bit- zur 64-Bit-Architektur Zeiger und den Typ `long` auf 64 Bit verbreitert und `int` weiterhin auf 32 Bit belassen. Dies wird kurz auch mit LP64 abgekürzt.

```c
#include <stdio.h>
#include <limits.h>   /* INT_MIN und INT_MAX */

int main(void) {
   printf("int Größe : %d Byte\n", sizeof( int ) );
   printf("Wertebereich von %d bis %d\n", INT_MIN, INT_MAX);
   return 0;
}
```

1.4.2 Variablen verwenden

```c
#include <stdio.h>

int main(void) {
   int a;    // Deklaration
   int b;
   int c;
   a = 5;    // Initialisieren
   b = 100;
   c = 12345;
   printf("Wert von int a=%d ,b=%d, c=%d\n", a, b, c);
   return 0;
}

int wert  = 5;                  // wert=5
int wert1 = 10, wert2 = 20;     // wert1=10 ,wert2=20

// wert1=nicht initialisiert, wert2=33
int wert1, wert2 = 33;

int wert1;
int wert2 = wert1 = 10;         // wert1=10, wert2=10
```

1.4.3 Long Integer (long)

4 Byte `long`: –2147483648 +2147483647

Der Datentyp `long` entspricht wie der Datentyp int auch einer Ganzzahlvariablen.

- Auf 16 Bit Systemen hat ein `long` 4 Byte also 32 Bit
- Auf 32 Bit Sytsmen hat ein `long` 4 Byte also 32 Bit
- Auf 64 Bit Systemeh hat wein `long` 8 Byte also 64 Bit

welche Daseinsberechtigung hat der Datentyp `long` dann eigentlich noch auf 32-Bit-Systemen?

- Es gibt ihn aus Kompatibilitätsgründen, damit alte Programme, die für 16-Bit-Rechner geschrieben wurden, auch noch auf einem 32-Bit-Rechner laufen bzw. übersetzt werden können

1.4.3 weitere Ganzzahl Typen

- `long long` ist ein 64 Bit (8 Byte) breiter Datentyp, der einen Wertebereich von –9.223.372.036.854.755.808 bis +9.223.372.036.854.755.807 darstellen kann.
- `short` ist ein 16 Bit (2 Byte) breiter Datentyp, der einen Wertebereich von –32768 bis +32767 darstellen kann.
- `char` ist ein 8 Bit (1 Byte) breiter Datentyp, der einen Wertebereich von –128 bis +127 (o bis 255) darstellen kann. Ein `char` kann auch zur Darstellung von einzelnen Zeichnen verwendet werden. ‘a‘, ‘b‘, ‘\n‘

1.4.4 Gleitpunkttypen

Bei Gleitpunkttypen wird auch von Zahlen mit gebrochenem Anteil (reellen Zahlen) gesprochen. Der C-Standard schreibt hierbei nicht vor, wie die interne Darstellung von reellen Gleitpunktzahlen erfolgen muss. Dies hängt von den Entwicklern der Compiler ab. Meistens wird aber der verwendet (IEEE – Institute of Electrical and EIEEE-Standard 754 lectronics Engineers).

```c
#include <stdio.h>

int main(void) {
   float f = 5.0;
   int i = 2;
   printf("%f\n",f/i); // Ergebnis = 2.500000
   return 0;
}
```

1.4.4 Gleitpunkttypen

Typ	Bitbreite	Wertebereich	Genauigkeit
`float`	4 Byte	1.2E-38 3.4E+38	6-stellig
`double`	8 Byte	2.3E-308 1.7E+308	15-stellig
`long double`	10 Byte (abhängig vom Compiler)	3.4E-4932 1.1E+4932	19-stellig

1.4.4 Genauigkeit von Gleitpunkttypen

Eine Fließkommazahl mit 6-stelliger Genauigkeit wie float kann sechs Dezimalstellen (Nachkommastellen) nicht immer korrekt unterscheiden. Wenn beispielsweise die Zahl vor dem Komma (z. B. »1234,1234«) bereits vier Stellen besitzt, so kann sie nach dem Komma nur noch zwei Stellen unterscheiden.

Somit wären die Gleitpunktzahlen

1234,12345 und
1234,123999

als float-Zahlen für den Computer nicht voneinander zu unterscheiden. Mit 6-stelliger Genauigkeit sind die signifikanten Stellen von links nach rechts gemeint.

1.4.4 Genauigkeit von Gleitpunkttypen

```c
#include <stdio.h>

int main(void) {
   float x=1.1234;
   float dollar=100000.12;
   float end_float;

   double y=1.1234;
   double DOLLAR=100000.12;
   double end_double;

   printf("%f Euro mit float\n",end_float=dollar*x);
   printf("%f Euro mit double\n",end_double=DOLLAR*y);
   return 0;
}
```

1.4.5 Der Typ »void«

Der Typ `void` ist kein echter Datentyp und wird überall dort verwendet, wo kein Wert benötigt wird oder vorhanden ist. Bei Funktionen wird `void` verwendet, wenn eine Funktion keinen Wert zurückgibt oder die Funktion keinen Parameter hat

Das andere Anwendungsgebiet von `void` sind typenlose Zeiger bzw. genauer gesagt `void-Zeiger` (void *). Solche Zeiger werden häufig bei der Deklaration von Funktionen verwendet,

1.4.6 Ganzzahlkonstanten

Ganzzahlige Konstanten können als Dezimalzahl, als Oktalzahl oder als Hexadezimalzahl dargestellt werden.

dezimal	hexadezimal	oktal	Typ
30	0x1e	036	`int`
30U	0x1eu	036U	`unsigned int`
30L	0x1eL	036l	`long`
30ul	0x1EUL	036UL	`unsigned long`
30ll	0x1ell	036ll	`long long`
30ull	0x1eull	036ull	`unsigned long long`

1.5 Ausdrücke und Operatoren

Motivation:

Ausdrücke bestehen aus Operanden und Operatoren. Operanden sind Variable, Konstanten und Ausdrücke. In diesem Abschnitt werden die grundlegenden Operatoren von C kurz vorgestellt.

1.5.1 Wertzuweisung

Die Wertzuweisung ist ein Ausdruck. Sie stellt die übliche Methode dar, einer Variablen einen bestimmten Wert zuzuordnen.

- allg.: lvalue = ausdruck

lvalue ist ein modifizierbarer Speicherbereich, z.B. eine Variable oder ein Feldelement. Konstanten sind keine lvalue's.

- `x = 0`
- `y = x + 5`
- `a = b = 1` `/* entspricht a = ( b = 1) */`
- `c = getchar()`

Im Gegensatz zu anderen Programmiersprachen besitzt Wertzuweisung in C selbst einen Wert. Damit sind sogenannte Mehrfachzuweisungen erlaubt und üblich:

- `a = b = c = 1`

1.5.2 Arithmetische Operatoren

- `*` Multiplikation `a * b`
- `/` Division (bei int ganzzahliger Anteil `b / 2`
- `%` Rest der ganzzahligen Division (int) `c % 2`
- `+` Addition `a + b`
- `-` Subtraktion (zwei Operanden / binärer Operator) `a - 5`
- `-` negatives Vorzeichen (ein Operand / unärer Operator) `-a`

1.5.2 Dividieren von Ganzzahlen

Wenn zwei Ganzzahlen wie z. B. 4/3 dividiert werden, bekommen Sie als Ergebnis 1 zurück. Der Grund ist ganz einfach, denn der Datentyp int entspricht einer Ganzzahlvariablen und schneidet daher den nicht ganzzahligen Rest einfach ab.

Wird der Rest benötigt, können Sie den Modulo-Operator verwenden. Der Modulo-Operator hat % als Zeichen.

```
int main(void) {
    int x=5;
    int y=2;
    x=x%y;
    printf("Der Rest von 5/2=%d\n",x); /*Rest=1 */
    return 0;
}
```

1.5.3 Vergleichsoperatoren

- `<` kleiner als `a < 2`
- `>` größer als `b > 3`
- `<=` kleiner oder gleich `b <= 3`
- `>=` größer oder gleich `(x-y) >= 0`
- `==` gleich(1) `c == '\n'`
- `!=` ungleich `c != '\t'`

Laut Definition besitzt ein Vergleichsausdruck den Wert 1 (TRUE), wenn die durch den Vergleichsoperator spezifizierte Bedingung erfüllt ist, ansonsten den Wert 0 (FALSE).

1.5.3 Vergleichsoperatoren

Beispiel

```
int x, y, schalter;
x=4; y=2;
schalter = x > y ; /* schalter=1 da Bedingung erfüllt - TRUE */
```

Vorsicht: In Vergleichsausdrücken Unterschied von == und = beachten !

```
if ( a == 1 ) ... /* TRUE wenn a den Wert 1 besitzt */
if ( a = 1 ) ... /* immer TRUE */
if ( a = 0 ) ... /* immer FALSE */
if ( a = 2 ) ... /* immer TRUE */
```

Die letzten drei Ausdrücke sind in C zulässig, führen aber leicht zu Verwechslungen mit der 1. Form !

1.5.4 Logische Operatoren

- Es gibt logische Operatoren zum Vergleich von Ausdrücken

&&	bedingtes logisches UND	a<2 && b>0
||	bedingtes logisches ODER	c ==' ' || c ==','

- und zur logischen Negation eines Ausdruckes

!	logische Negation	! b

```
int x, y, z;
x = 3; y = 5;     //(wobei 3 = 011 und 5 = 101 ist)
z = x && y;       //==> z = 1       (1)
z = x || y;       //==> z = 1       (1)
```

1.6 Ablaufsteuerung

Motivation:

In diesem Abschnitt werden nur die grundlegenden Möglichkeiten zur Ablaufsteuerung in C dargestellt, die mit anderen Programmiersprachen unmittelbar korrespondieren.

1.6.1 Einfache Anweisung

Anweisungen legen die vom Programm auszuführenden Aktionen fest. Eine einfache Anweisung ist ein Ausdruck, dem ein Semikolon ; folgt:

```
ausdruck ;
a = b + c ;
printf("Mein C-Programm.\n") ;
;                          ===> leere Anweisung
```

Jeder Ausdruck besitzt einen Wert.

Anweisungen besitzen keinen Wert (aber die in der Anweisung enthaltenen Ausdrücke).

In einer Funktion werden die Anweisungen sequentiell abgearbeitet.

Mit Steuerstrukturen (bedingte Anweisungen, Schleifen, ...) kann die Abarbeitungsfolge beeinflusst werden.

1.6.2 Anweisungsblock

- mit Hilfe geschweifter Klammern { } werden Vereinbarungen und Anweisungen zu einem Block zusammengefasst
- an jeder Stelle, wo eine Anweisung stehen darf, kann auch ein Block stehen
- der schließenden geschweiften Klammer folgt kein Semikolon
- Blöcke können beliebig geschachtelt werden
- auf eine im Block vereinbarte Variable kann nur innerhalb des Blocks zugegriffen werden

```
main()
{                       /* Beginn Block */
   int i,n;             /* Vereinbarungen */
   ...
   i = 0;               /* Anweisungen */
   n = i = 1;
   ...
}                       /* Ende Block */
```

1.6.3 if-else-Anweisung

if (*ausdruck*)
  anweisung$_1$

else *
  anweisung$_2$

* Der else-Zweig gehört zum unmittelbar vorhergehenden if und ist optional.

- *ausdruck* ist TRUE, wenn Wert != 0 (beliebig!)
- *ausdruck* ist FALSE, wenn Wert == 0 .

```
#include <stdio.h>

main()
{
   int x, y, z;

   x = 3; z = 2;
   if ( z != 0 )     /*  auch:   if ( z )  */
      y = x/z;
   else {
      printf("Division durch Null\n");
      y=0;
   }
   printf("y = %d\n",y);
}
```

1.6.3 if-else-Anweisung

if (*ausdruck*)
    $anweisung_1$

else if
    $anweisung_2$

else if
    $anweisung_{n-1}$

else
    $anweisung_n$

```
#include <stdio.h>

main()
{
   int z;

   z = 1;
   if (z == 1)
      printf("z = %d\n",1);
   else if (z == 2)
        printf("z = %d\n",z = z + 1);
   else if (z == 0)
        printf("z = 0\n");
   else
        printf("z = %d\n",z);
}
```

1.6.4 swich case

Die Anweisung switch ist eine spezielle bedingte Anweisung. Allg.:

```
switch ( ausdruck) {
case konst_ausdruck1 :
      anweisungs_liste1
case konst_ausdruckᵢ :
      anweisungs_listeᵢ
default:
      anweisungs_listed
case konst_ausdruckᵢ₊₁ :
      anweisungs_listeᵢ₊₁
case konst_ausdruckn :
      anweisungs_listen
}
```

Der Wert von `ausdruck` wird mit den Werten von konst_`ausdruck`$_j$ (j = 1, 2, ...) verglichen. Bei Übereinstimmung wird die dazugehörige `anweisungs_liste`$_j$ abgearbeitet.

ACHTUNG! Nach Ausführung der `anweisungs_liste`$_j$ eines case-Zweiges wird die switch-Anweisung nicht automatisch beendet, sondern `anweisungs_liste`$_{j+1}$ usw. ausgeführt. Dies kann aber mit Hilfe der `break-Anweisung` verhindert werden.

1.6.5 while-Schleifen

Die Anweisung while ist eine Laufanweisung.

while (*ausdruck*)
    anweisung

```
#include <stdio.h>

main()
{
    int s, i;

    s = 0;  i = 1;
    while ( i <= 10 ) {
        s = s + i;
        i = i + 1;
    }
    printf("s = %d, i = %d\n", s, i);
}
```

Als erstes wird *ausdruck* ausgewertet. Bei einem Wahrheitswert TRUE (!=0) wird *anweisung* ausgeführt und anschließend der Vergleichsausdruck berechnet. Dies geschieht solange, bis *ausdruck* gleich FALSE (==0) ist, d.h. die while-Anweisung ist hier beendet.

1.6.6 do-while-Schleifen

do
    anweisung
while (*ausdruck*);

```
#include <stdio.h>

main()
{
    int s, i;

    s = 0; i = 99;
    do
        s = s + i;
    while ((i = i - 1) > 0);
    printf("s = %d, i = %d\n",s, i);
}
```

Im Gegensatz zur while-Konstruktion wird bei der do-while-Schleife *ausdruck* erst nach Ausführung von *anweisung* überprüft. Daraus folgt, dass diese Schleife mindestens einmal abgearbeitet wird.

1.6.7 for-Anweisung

Die Anweisung for ist eine spezielle Laufanweisung. Allg.:

for ($ausdruck_1$; $ausdruck_2$; $ausdruck_3$)
    anweisung

- $ausdruck_1$ ist Schleifeninitialisierung
- $ausdruck_2$ ist Wiederholungstest (Vergleichsausdruck)
- $ausdruck_3$ ist Wiederinitialisierung

Diese for-Anweisung entspricht folgender allg. while-Anweisung:

$ausdruck_1$;
while ($ausdruck_2$)
{
        anweisung;
        $ausdruck_3$;
}

1.6.7 for-Anweisung

Bei der for-Anweisung kann jeder beliebige ausdrucki weggelassen werden. Es gilt:

for (; ;)

ist eine unendliche Schleife.

Wenn in einer for-Schleife mehrere Laufvariablen benötigt werden, findet der Kommaoperator Anwendung:

```
#define N 100;
...
int i, j;
...
for(i=0,j=N ; i<N ; i++,j--) ..
```

1.6.8 Anweisungen zur unbedingten Steuerungsübergabe

Die break-Anweisung wird verwendet, um die Abarbeitung einer unmittelbar übergeordneten **switch-, while-, do-while-** bzw. **for-Anweisung** abzubrechen. (z.B.: Verlassen einer "unendlichen" for-Schleife [for (; ;)] mit break).

Eine continue-Anweisung bewirkt das Einleiten der nächsten Iteration der umgebenden Schleife.

Desweiteren kann man die Anweisung goto marke ; verwenden. Dabei wird mit marke : das Sprungziel einer goto-Anweisung markiert. Jedes Programm kann ohne goto geschrieben werden; es ist eigentlich nur sinnvoll zum schnellen Verlassen mehrfach geschachtelter Blöcke.