Noch einmal Organisatorisches

- aktuelle Informationen: Webseite
  http://www.stud.informatik.uni-goettingen.de/c-kurs/ss2007/#termine
- email-Kommunikation: studIP (email-Weiterleitung!)
- Klausur: Registrierung
- Termine: täglich Vorlesung und Übungen

- Übungszettel: Standardaufgaben, Ergänzungsaufgaben
- Literatur

- Kursziel / Schein

Übersicht

- Interne Zahlendarstellung
- Standarddatentypen
- Konstanten und Variablen
- Bezeichner
- Ausdrücke
- Operatoren
- Formatierte Ein- und Ausgabe
- Variablen und Adressen
- Felder
- Spezielle Ein- und Ausgabefunktionen

- 109 <=> 1101101

- Dezimaldarstellung von 109:
  Zerlegung in die 10er Stellen
  9 * 10^0 + 0 * 10^1 + 1 * 10^2

- Interne Abbildung durch 0 und 1 (Binärdarstellung)

  $(x_0 * 2^0) + (x_1 * 2^1) + (x_2 * 2^2) + (x_3 * 2^3) \ldots$
  $= (x_0 * 1) + (x_1 * 2) + (x_2 * 4) + (x_3 * 8) \ldots$

  109 = 64+32+0+8+4+0+1
  -> 1101101

[Datentypen] Bits, Bytes

- 1 Bit speichert genau die Information 0 oder 1
  (rechnerintern: es fließt ein Strom oder nicht)
- 1 Byte: 8 Bits
  2^8 Zahlen mit einem Byte darstellbar
  0 .. 255 (ganzzahlig, nichtnegativ)
  -127 .. 127 (ganzzahlig, 1 Stelle für das Vorzeichen)

- INT_MAX = 2 147 483 647
  INT_MIN = -2 147 483 648
- 4294836226 = 2^32 -> 32 Bits bzw. 4 Bytes werden zur Darstellung benötigt.
- 109 = 00000000 00000000 00000000 01101101

Exkurs: 2er Komplement

- Möglichkeit 1: eine Stelle für das Vorzeichen
  Nachteil: + und – 0
- Möglichkeit 2: 2er Komplement

  Bei einer Wortgröße 2^n folgt:
  Positiver Zahlenbereich 0 .. (2^(n-1) -1)
  Negativer Zahlenbereich – 2^(n-1) .. 0

- Beispiel: Wortgröße 2^8 <-128 .. 127>

```
-128: 10000000
-127: 10000001
   -1: 11111111
    0: 00000000
    1: 00000001
  127: 01111111
```

[Datentypen] Standard Datentypen 1

- Literale

109 0155 0x6D	int: dezimal, oktal, hex
'a' '\t' “string“	a character, tabulator, string
3.14f 3.14L 3.14E+0	float, long double, double (default)

- Datentypen:
  - Jeder Datentyp hat einen Wertebereich, plattformabhängig
  - Integer: (unsigned) short int, int, long int
  - Zeichen: char
    intern eine Teilmenge der ganzen Zahlen, belegt ein Byte
    0..31 , 127 -> Steuerzeichen
    32 .. 126 -> druckbare Zeichen

[Datentypen] char und int

- char ist eine Teilmenge von int
- es gibt eine vorgeschriebene Menge an Zeichen, welche auf allen Rechnern darstellbar sein müssen (128 Zeichen des ASCII-Codes)
  -> Interpretation eines Zahlwertes durch ein Zeichen (einer Konstante) laut Tabelle

```
char c = 69;
int i = 'P';     /* ist auch gültig */

i = 'E';         /* Zuweisung einer Konstanten */

if ('E' == 69)       /*  Vergleich von Konstanten und Zahlwert ist gültig */
{
        printf("%d", i);          /* unterschiedliche Formatbeschreiber */
        printf("%c", i);          /* (wird später beschrieben) */
}

'a' + 5;         /* gültiger Ausdruck */
```

[Datentypen] char und int

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

[Datentypen] Gleitkomma-Typen

- Gleitkommatypen:
  float, double, long double

  Definiert durch

  - den Wertebereich (z.B. FLT_MIN,FLT_MAX)
  - die Dichte (= Abstand von einer darstellbaren Zahl zur nächsten) (z.B. FLT_EPSILON)
  - Anzahl der signifikanten Stellen (z.B. FLT_DIG)

[Datentypen] Datentypen und Wertebereiche

- Wertebereiche der Datentypen sind in <limits.h> und <float.h> festgelegt.

```
SHRT_MIN, SHRT_MAX, USHRT_MAX
INT_MIN, INT_MAX, UINT_MAX
LONG_MIN, LONG_MAX, ULONG_MAX

CHAR_MIN, CHAR_MAX
SCHAR_MIN, SCHAR_MAX
UCHAR_MAX

FLT_MAX, DBL_MAX, LDBL_MAX                (Wertebereich)
FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON    (Dichte)
FLT_DIG, DBL_DIG, LDBL_DIG                (Anzahl signifikante Stellen)
```

```
#include <limits.h>
#include <float.h>
...
printf("Kleinster darstellbarer Integer-Wert: %d\n",INT_MIN);
printf("Groesster darstellbarer Integer-Wert: %d\n",INT_MAX);
printf("Kleinstmoeglicher Abstand zweier Float-Zahlen: %f\n",FLT_EPSILON);
printf("Groesstmoegliche Float-Zahl: %f\n",FLT_MAX);
```

[IO] Formatierte Ausgabe

- printf – Funktion benötigt Typ-Information, wenn sie mehr als den bisher bekannten String darstellen soll.

```
printf(“Hallo Welt\n“);

int jahr = 2007;
printf(“Wir schreiben das Jahr: “,jahr);  /* Alarm! so gehts nicht */
```

- Obige Variante funktioniert so nicht. Die Funktion kennt den Typ der Variablen nicht, er muss ihr mitgeteilt werden. Dies geschieht per Konvention mit Formatbeschreibern (einer pro Variable):

```
printf(“Wir schreiben das Jahr: %d\n“,jahr);

int geburtsjahr = 1971;
printf(“Jahr: %d  mein Alter: %d“,jahr , (jahr-geburtsjahr));
```

[IO] Formatierte Ausgabe

- Für jeden Datentyp gibt es einen Formatbeschreiber (oder einen zusammengesetzten Formatbeschreiber)

```
i,d     signed int
u,o,x,X unsigned int
f       double (exponentenfrei)
e E     double (halblogarithmisch)
g G     double (exponentenfrei oder halblogarithmisch)
c       char
s       Zeichenfolge (char *)
p       Adresse (void*)
```

- kein Beschreiber für float (wird nach double gerechnet) ebenso bei short (Umwandlung nach int)
- für long-Werte wird ein l vorangestellt

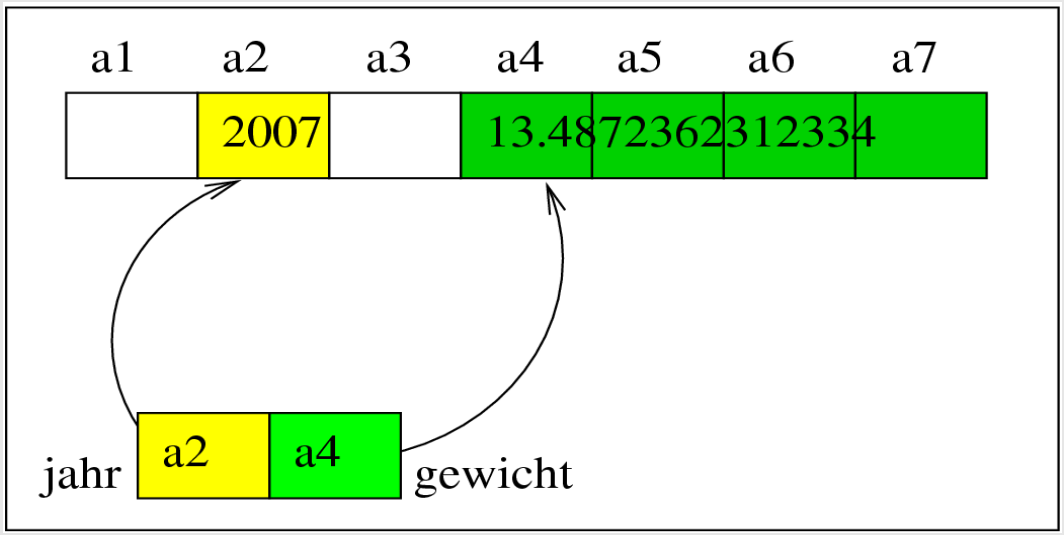
[IO] Variablen und Adressen

- Vorbemerkung: Adressen

```
printf("Wir schreiben das Jahr: %d\n",jahr);

printf("Jahr: %d  Adresse der jahr-Variablen:%p\n", jahr, &jahr);
```

- Variablen haben einen Speicherplatz und einen Wert. Der Speicherplatz ist die Adresse (vom Typ unsigned int)
- An dieser Adresse ist der Wert gespeichert



[IO] Einschub: Variablen und Adressen

- Eine Variable repräsentiert eine Adresse (Beschreiber)
- Wird eine Variable benutzt, so wird die Adresse automatisch „dereferenziert“ auf den Wert, auf welchen die Adresse zeigt.
- Will man tatsächlich die Adresse, muss man die Dereferenzierung unterbinden:
  Adressoperator &

```
printf(“Das ist die Adresse der Variablen “
          “aktuelles_jahr: %p \n“, &aktuelles_jahr);
```

[Datentypen] Typumwandlungen

- Es können in arithmetischen Operationen zwar alle Typen miteinander verbunden werden, der Rechner kann aber nur Operanden gleichen Typs verknüpfen.
- geschieht automatisch: implizite Typumwandlung

```
Typhierarchie:

long double
double
float
unsigned int / long
int / long
```

- Dabei wird der niederwertigere Typ in den höherwertigen umgewandelt

```
float pi = 3.14; int r=4; float a;
a = r*r*pi = 16 * 3.14 = 16.0 * 3.14
```

[Datentypen] Implizite Typumwandlungen, Frts.

- Das verwenden von 'falschen' Formatbeschreibern kann zu unerwarteten Ergebnissen führen

  (imlizite Typumwandlung bei der Multiplikation eines int mit einem unsigned int, keine Typumwandlung bei der Ausgabe eines unsigned int als int durch printf)

```
unsigned int ui = 10;
int i = -1;

printf("Wir multiplizieren %u mit %d, Ergebnis: %u %d \n"
         , ui , i, ui * i, ui * i );

/*
ui * i: 4294967286
ui * i: -10
*/
```

[Datentypen] Typumwandlungen, Typematching

- Explizite Typumwandlung: Typecast

```
printf(“Quotient: %d“,z/n);                 /* Gefahr von div. durch 0 */

printf(“Quotient: %d“, (double)z/n          /* Umwandlung von z in einen double
                                               Wert. Durch impliziten Typecast
                                               wird auch n umgewadelt */
```

- Formatbeschreiber und Typ müssen bei der Ein- und Ausgabe übereinstimmen. Sorgt der Benutzer nicht dafür, kann es zu unerwarteten Ergebnissen kommen

```
printf(“Eine Gleitkommazahl: %f\n“,3.0234);

printf(“Eine Gleitkommazahl: %d\n“,3.0234);

printf(“Eine Gleitkommazahl: %d\n“,(int)3.0234);
```

[IO] Formatierte Eingabe

- Funktion scanf()
- Auch diese Funktion benötigt Parameter: die *Adresse* der Variablen, in welche die Eingabe geschrieben werden soll:

```
int aktuelles_jahr, etwasanderes;

scanf(“%d“, &aktuelles_jahr);
scanf(“%d %d“,&aktuelles_jahr,&etwasanderes)
```

- 2. Parameter: *Formatbeschreiber*
- Das wird nicht funktionieren:

```
scanf(“%d“, aktuelles_jahr);  /*  es wird ein wert statt einer Adresse
                                  übergeben */
scanf(“Jahr eingeben: %d“, &aktuelles_jahr);
                              /*  der erste Parameter, welcher die
                                  Formatbeschreiber enthält, sollte nur
                                  diese enthalten, keinen Text */
```

[Datentypen] Konstanten

- 'a' eine Zeichenkonstante. Sie repräsentiert die Ordnungszahl ihres Zeichens.

- Strings = Zeichenkettenkonstanten
  “Dies ist ein String. Leerzeichen, \t (Tabulator) und “
  “ newline sind sogenannte whitespaces, ein backslash \n“
  “leitet eine Escapesequence ein. Ein backslash “
  “allein ist nicht erlaub, so \\ aber schon. “

  Ein String muss vor dem Zeilenende abgeschlossen werden, kann dann aber weitergeführt werden.

  Der String “ach!“ intern: ['a'|'c'|'h'|'!'|'\0'] -> 5 Zeichen

Einschub: Escape-Sequenzen

```
\a  nervigste alle Escapesequenzen
\b  Backspace
\f  Seitenvorschub (nur Drucker)
\n  Zeilenvorschub
\r  Positionierung auf Zeilenanfang
\t  horizontaler Tabulator
\v  vertikaler Tabulator (nur Drucker)

\'  Apostroph
\“  Gänsefüßchen
\\  Backslash
\?  Fragezeichen (bei zwei aufeinanderfolgenden Fragezeichen)
```

[Datentypen] Konstanten Forts.

- Aufzählungskonstanten: enum
  enum <name> { konstantenliste};

```
enum wochentage
     {MONTAG,DIENSTAG,MITTWOCH,DONNERSTAG,
          FREITAG, SAMSTAG,SONNTAG};

int wochenanfang = MONTAG ;

enum wochentagemitwerten
{ MONTAG = -1, DIENSTAG = -2, MITTWOCH = -3, DONNERSTAG = -4 ,
    FREITAG = 0 , SAMSTAG = 1, SONNTAG = 2};
```

- Benannte Konstante mit #define

  #define name ersatztext

```
#define MYAGE 35

printf(“Mein Alter: %d “, MYAGE);
printf(“Mein Alter: MYAGE“);
```

[Datentypen] Variablen

- Definition:
  Typ Variablenname [Wert];
- Definitionen erfolgen immer
  - am Anfang des Programms (global)
  - am Anfang einer Funktion

```
int main(){
    int seitenzahl = 1;   /* Definition mit Initialisierungswert */

    /* später im Programm */

    seitenzahl = 2;       /* neue Wertzuweisung */
    ...
```

Einschub: Deklaration vs. Definition

- Jede Definition ist auch eine Deklaration
- Bei einer Definition wird tatsächlich einer Variable Speicherplatz zugewiesen. Dies kann genau einmal geschehen
- Bei einer Deklaration wir lediglich ein Bezeichner und sein Typ „bekannt gemacht“. Dies kann mehrmals geschehen, wie wir später im Kurs noch sehen werden.

[Datentypen] Bezeichner

- Bezeichner (Namen) zur Benennung von Konstanten, Variablen, Funktionen
- erlaubt sind alphanumerische Zeichen und _
  { [a,z] , [A,Z] , [0,9] ,'_' }
- Das erste Zeichen eines Namens darf keine Ziffer sein
- Nicht verwendet werden dürfen folgende Schlüsselwörter der Sprache C (case-sensitiv)

```
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
```

[Datentypen] Typnamen

- mit `typedef` können eigene Namen für Typen definiert werden.

```
typedef int meinint_t;
int testzahl;

meinint_t alter = 99;
testzahl = alter;
```

```
enum wochentage { MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG,
                  SAMSTAG, SONNTAG };

typedef enum wochentage tage_t;   /* tage_t ist ein neuer Typ */
tage_t wochenanfang;              /* wochenanfang ist eine neue  */
tage_t klausurtag;                /* Variable vom Typ tage_t /
wochenanfang = MONTAG;
klausurtag = FREITAG;
printf("%d %d",wochenanfang, klausurtag);
                                  /* Was kommt dabei heraus ? */
```

Gute Bezeichner

- Konventionen für Bezeichner
  - #define KONSTANTEN_GROSSSCHREIBEN
  - int variablen_kleinschreiben
  - int a = 35; /* Bezeichner sollten selbst */
    int alter = 35; /* für sich sprechen können */
    int mein_alter = 35;
  - int variable_fuer_altersangaben;
    int womoeglicheinetwaslangervariablenname;

[Datentypen] Ausdrücke

- Ausdrücke bestehen aus
  - Operanden: Konstante, Variablen, Funktionen
  - Operatoren
  - Klammern
- ... und werden mit Bildungsregeln interpretiert
  - Prioritäten von Klammern und Operatoren
- Ausdrücke haben einen Wert

```
i - 1
(a + b) -7
i < 13
i = j + 3 * ( k + 2 )
i = ( k + 3 , 5) ;        /*  i = ?   */
```

[Datentypen] Zuweisungen

- Zuweisungen sind auch Ausdrücke
  (und keine Anweisungen)

```
d = a + b ;       /* das Semikolon macht den Ausdruck zur Anweisung */

if(d = a + b)     /* was einerseits eine Zuweisung ist, ist hier auch ein
                     Ausdruck */
```

- Der Wert des gesamten Ausdrucks entspricht dem Wert des Ausdrucks rechts des Zuweisungsoperators
- Links des Zuweisungsoperators dürfen nur Ausdrücke stehen, welche einen Speicherplatz haben (an welche das Ergebnis geschrieben werden kann).
- Später werden wir sehen, dass es formal richtig heissen müßte:
  &d = a + b;
- siehe auch Nebeneffekte

- Arithmetik

  a +b
  10 % 3
  10.0 / 3
  10 / 3

```
+
-
/
*
%       Modulo
--      Decrement (pre/post)
++      Increment (pre/post
```

- Wertzuweisung

  i = i + 1
  i += 1
  j = ++i equ. i=i+1 ; j=i
  j = i++ equ. j=i ; i=i+1

```
=
*=      Multipliziere
/=      Dividiere.
%=      Modulus.
+=      Addiere.
-=      Subtrahiere. (=- ist etwas anderes)


<<=     Linksverschiebung.
>>=     Rechtsverschiebung.
&=      Bitweises UND (AND).
^=      Bitweises Exklusiv-ODER (XOR).
|=      Bitweises Inclusiv-ODER OR).
```

- Relational / Logisch
  Ausdrücke mit diesen Operatoren haben entweder den Wert 0 oder 1.
  (Es gibt keinen boolschen Datentyp mit Werten true oder false).

```
==  gleich
!=  ungleich
>   größer als
<   kleiner als
>=  größer gleich
<=  kleiner gleich
&&  logisches UND (AND)
||  logisches ODER (OR)
!   Negation (NOT)
```

```
a = 1; b = 1; c = 0; d = 0;

a == b;      /* 1 entspr. true */
a == c;      /* 0 entspr. false */
a != c;      /* 1 */
a || c;      /* 1 */
c || d;      /* 0 */
a && b;      /* 1 */
a && c;      /* 0 */
!a           /* 0 */
```

[Datentypen] Priorität der Operatoren

Prioritt	Operator	Assoziativität
15	( ) [ ] -> .	--->
14	! ~ ++ -- + - (TYP) * & sizeof	<---
13	* / % (Rechenoperationen)	--->
12	+ - (binär)	--->
11	<< >>	--->
10	< <= > >=	--->
9	== !=	--->
8	&	--->
7	^	--->
6	|	--->
5	&&	--->
4	||	--->
3	?:	<---
2	= += -= /= *= %= >>= <<= &= |=	<---
1	, (Sequenz-Operator)	--->

- Assoziativität:
  Ausdrücke gleicher Priorität werden gemäß ihrer Assoziativität abgearbeitet.

```
Rechtsassoziativ
(von rechts nach links)

a = b = c = d = 1;

a = (b = (c = ( d = 1 )))



Linksassoziativ
(von links nach rechts)

5 – 1 + 10

( 5 – 1 ) + 10
```

Besondere Ausdrücke

- Bedingte Ausdrücke

```
bedingung  ?  ausdruck1  :  ausdruck2
```

wenn der Ausdruck *bedingung* wahr ist (1), dann wird der Ausdruck *ausdruck1* zum Wert des Gesamtausdrucks.
Ist aber der Wert der Bedingung falsch (0), dann wird *ausdruck2* zum Wert des Gesamtausdrucks.

```
a = 5; b = 7;
maximum = a > b ? a : b ;
```

- Kommaoperator
  Mehrere Ausdrücke können durch *komma* getrennt in eine Zeile geschrieben werden.

```
h=x , x=y , y=h;      /* der Wert des Gesamtausdrucks entspricht
                         dem des zuletzt ausgewerteten
                         Teilausdrucks */
```

[IO] Einschub: while-Schleifen

- Formal: Wiederholte Ausführung von Anweisungen

```
while ( Bedingung )
{

    anweisungen ;

}
```

Die Anweisungsfolge wird solange wiederholt, solange die Bedingung *wahr* ist.

```
while ( 1 ) { printf("C ist toll"); }     /* Endlosschleife */

while ( 0 ) { printf("Ich mag kein C");}  /* Nimmer-Schleife */

int a = 0;
while ( a != 1 ){a++;}                    /* Einmal-Schleife */
```

[IO] Zeilenende: EOF

- EOF: EndOfFile (das Ende einer Eingabezeile)
- Rückgabewert von scanf, wenn control-d gedrückt wurde

```
#include <stdio.h>
...
printf("EOF: %d ",EOF);   /* meistens -1 */
```

- Läßt sich als Schleifenbedingung verwenden:

```
while ( scanf("%d", &alter) != EOF)
{
    /* hier passiert was*/
}
```

Solange eine Zahl eingegeben wird, ist der Rückgabewert von scanf > 0, die Schleife wird nochmal durchlaufen.

[IO] Funktionen zur Ein- und Ausgabe

- getchar() , putchar(char)

```
while ((c = getchar()) != EOF)
{
    putchar(c);
}
```

- getchar() liest Text von der Standardeingabe; hier Tastatur, aber auch andere Eingabewege möglich (Demonstration von pipes in Unix):

```
echo Ein kleiner Text | meingetcharprogramm
```

[IO] Felder (arrays)

- Datenstruktur zur Darstellung von Vektoren, Matrizen, Tabellen
  (1-dimensionale Felder = Vektoren
  2-dimensionale Felder = Matrizen)
- Ein Feld besteht aus einer fixen Anzahl von Elementen gleichen Typs.
- Bei der Initialisierung gibt der Wert in der Klammer die Größe des Feldes an (=Anzahl Komponenten):

```
#define FELD_LAENGE  5
int mein_vektor[FELD_LAENGE];
int initialisierter_vektor[] = { 1 , 2 , 3 , 4 , 5 };
```

- Adressierung einer Komponente:

```
mein_vektor[0]                /* erstes Element startet bei 0 !! */
mein_vektor[FELD_LAENGE]      /* liegt ausserhalb des Feldes ! */
mein_vektor[FELD_LAENGE-1]    /* Letztes Element des Feldes */
```

[IO] Felder und Schleifen

- Schleifen eignen sich hervorragend, um mit Feldern zu arbeiten

```
#define LAENGE 10
int mein_vektor[LAENGE];
int schleifenvariable = 0;

/* Abbruch mit control-d möglich */
while(scanf(“%d“, &mein_vektor[schleifenvariable]) > 1 &&
        schleifenvariable < LAENGE)
{
    schleifenvariable++;
}

schleifenvariable = 0;

/* Ausgabe der eingegebenen Komponenten des Feldes */
while(schleifenvariable<LAENGE)
{
    printf("%d ",mein_vektor[schleifenvariable++]);
}
```

- Uninitialisierte Feldkomponenten enthalten „Zufallswerte“

[IO] Felder, Zeichen und Strings

- Eine Stringvariable wird mit einem char[] (Feld von char-Werten) realisiert.

```
#define LAENGE 10

char buchstabe = 'A';
char buchstabe = “A“;          /* geht nicht, das wäre eine
                                  String-Initialisierung */
char text[9]=“Mein Text“;      /* geht */
char text[] =“Mein Text“;      /* geht */
char text[8]=“Mein Text“;      /* geht nicht, Feld zu klein */

char text[] = {'M','e','i','n',' ','T','e','x','t','\0' };
```

- Intern wird ein String mit '\0' (ASCII-Null) beendet.

```
schleifenvariable = 0;
while (text[schleifenvariable] != '\0')
{
    printf(“%c“,text[schleifenvariable++]);
}
```

[IO] Zeichenklassifizierung und Stringbehandlung

```
<ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

```
<string.h>

char *strcat(char *s1, const char *s2);
int strcmp(const char *s1, const char *s2);
char *strcpy(char *s1, const char *s2);

char *strncat(char *s1, const char *s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);

size_t strlen(const char *s);

[....]
```

[IO] Stringbehandlung: Ein Beispiel

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

int main()
{
    char string1[100];
    char string2[] = "Teil 2 angehaengt.";
    printf("%d\n", strlen(string2));

    printf("%d\n", strlen(string1));
    strcpy(string1,"Teil 1 kopiert. ");
    printf("%d\n", strlen(string1));

    strcat(string1,string2);
    printf("%d\n", strlen(string1));

    printf("%s\n", string1);

    return 0;
    }
```

[IO] Beispiel mit einer ctype-Funktion

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

int main()
{
    int c;
    unsigned int x;

    x = 0;

    while (isxdigit( c = getchar()))
    {
        x = x * 16 + (isdigit(c) ? c – '0' : toupper(c) – 'A' + 10);
    }
    printf("%d\n",x);
    return 0;
}
```

[Datentypen] sizeof Operator

- Der C-Standard verlangt: sizeof(char) == 1
- Das Resultat von sizeof() besitzt den Typ size_t
- Kein Funktionsaufruf! Keine Nebeneffekte!

```
size_t s1,s2,s3;
int i;
char c;
char feld[5];

s1 = sizeof(i);
s1 = sizeof(c);
s3 = sizeof(size_t);

printf("Größe (int): %d \t (char): %d\t (sizeof): %d\n", s1, s2, s3);
printf("Größe (float): %d \t %d \n", sizeof(4.0), sizeof(float));


/* Auch Funktionen haben eine Größe (entsprechend ihres Typs) */
printf("Größe von main(): %d\n", sizeof( main() ) );


/* Grösse eines Feldes */
printf("Größe des Feldes von char-Werten: %d\n",sizeof(feld));
```