6 Bibliotheksfunktionen

Einige Funktionen aus time.h	
Funktion	Aktion
`clock_t CLOCKS_PER_SEC`	Variable mit tics pro Sekunde
`clock_t clock(void)`	Rechenkernzeit in tics
`time_t time(time_t *pt)`	aktuelle Kalenderzeit oder -1

Einige Funktionen aus stdlib.h	
Funktion	Aktion
`int RAND_MAX`	Variable mit maximaler Zufallszahl
`int abs(int)`	$\|i\|$
`void abort(void)`	Abbruch des Programms
`long labs(long)`	$\|l\|$
`int rand(void)`	positive ganze Zufallszahl
`void srand(unsigned)`	Zufallsgenerator initialisieren

Listing 23: Zufallszahlen (random.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
        time_t g;
        /* Systemzeit */
        g = time(NULL); /* oder g =4711 */
        printf("Generator: %u\n", (unsigned) g);
        /* Initialisierung des Generators */
        srand( (unsigned) g);
        /* Zufallszahlen */
        printf("in [0,100]: %d\n", rand() % 101);
        printf("in [20,29]: %d\n", rand() % 10 + 20);
        printf("maximale Zufallszahl: %d\n", RAND_MAX);
        return 0;
}
```

Funktionen aus math.h	
Funktion	Aktion
`double acos(double)`	arccos(d)
`long double acosl(long double)`	
`double asin(double)`	arcsin(d)
`long double asinl(long double)`	
`double atan(double)`	arctan(d) in $[-\pi/2, \pi/2]$
`long double atanl(long double)`	
`long double atan2l(long double, long double)`	
`double atan2(double,double)`	arctan(d1/d2) in $[-\pi, \pi]$
`double ceil(double)`	auf nächste ganzen Zahl aufrunden
`long double ceill(long double)`	
`double cos(double)`	cos(d)
`long double cosl(long double)`	
`double cosh(double)`	cosh(d)
`long double coshl(long double)`	
`double exp(double)`	e^d
`long double expl(long double)`	
`double fabs(double)`	$\|d\|$
`long double fabsl(long double)`	
`double floor(double)`	auf nächste ganzen Zahl abrunden
`long double floorl(long double)`	
`double fmod(double,double)`	Rest von d_1/d_2 mit sign d_1
`long double fmodl(long double, long double)`	
`double frexp(double, *int)`	Mantisse und ganze Zweierpotenz
`long double frexpl(long double, int *)`	
`double ldexp(double, int)`	$d * 2^i$
`long double ldexpl(long double, int)`	
`double log(double)`	ln(d)
`long double logl(long double)`	
`double log10(double)`	log(d)
`long double log10l(long double)`	
`double modf(double, *double)`	Vor - und Nachkommaanteil von d
`long double modfl(long double, long double *)`	
`float modff(float, *float)`	Vor - und Nachkommaanteil von d

Funktionen aus math.h	
Funktion	Aktion
`double pow(double,double)`	$d_1^{d_2}$
`long double powl(long double, long double)`	
`double sin(double)`	sin(d)
`long double sinl(long double)`	
`double sinh(double)`	sinh(d)
`long double sinhl(long double)`	
`double sqrt(double)`	$\sqrt{d}$
`long double sqrtl(long double)`	
`double tan(double)`	tan(d)
`long double tanl(long double)`	
`double tanh(double)`	tanh(d)
`long double tanhl(long double)`	

Listing 24: Funktionen aus math.h (1) (mafunc1.c)

```
#include <stdio.h>
#include <math.h>
int main(void) {
        double x= 1949.1951, y;
        int expo;
        y = frexp(x, &expo);
        printf("%f, %f, %d\n", x, y, expo);
        return 0;
}
```

Output:

```
1949.195100, 0.951755, 11
```

Listing 25: Funktionen aus math.h (2) (mafunc2.c)

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

int main(void)
{
        double x= 1949.1951, y, z=0., a=-1.;
        x = modf(x,&y);
        printf("Vorkomma: %f, Nachkomma: %f\n", y, x);
        /* abzuraten ist von
         * printf("Nachkomma: %f, Vorkomma: %f\n",
         * modf(x,&y), y); */
        printf("0.^-1 = %f\n", pow(z,a));
        printf("0./0. = %f\n", z/z);
        printf("log(0) = %f\n", log(z));
        return 0;
}
```

```
Vorkomma: 1949.000000, Nachkomma: 0.195100
0.^-1 = -Inf
0./0. = NaN
log(0) = -Inf
```

6.1 Operatoren (3)

? : Operator:

expr1 ? expr2 : expr3 es wird *expr1* ausgewertet, ist das Ergebnis von 0 verschieden, wird *expr2* ausgewertet und zurück gegeben. Ist *expr1* nach Auswertung 0, so wird *expr3* ausgewertet und zurück gegeben.

Folgen Operator:

***wert* = (*expr1*, *expr2*, ..., *expr3*);** Die Ausdrücke werden von links nach rechts ausgewertet, *wert* erhält den Wert (und Typ) von *expr3*.

Listing 26: Euklidischer Algorithmus (euklid2.c)

```
#include <stdio.h>

static int ggt(int, int);

int main(void)
{
        int m, n, tmp;
        printf("ggT(n,m) berechnen\n");
        printf("Eingabe n, m: ");
        (void) scanf("%d %d", &n, &m);
        /* diesen Block braucht man nicht */
        if ( n > m ) {
                n = (tmp = m, m = n, tmp);
        }
        /* ..... */
        printf("ggT: %d\n", ggt(m, n));
        return 0;
}

int ggt(int m, int n)
{
        return (m % n) ? ggt(n, m % n) : n;
}
```

Funktionen aus stdio.h	
Funktion	Aktion
`int getchar()`	Zeichen von stdin lesen
`int printf(...)`	Ausgabe auf stdout
`int putchar(c)`	Ausgabe von c auf stdout
`int scanf(...)`	Einlesen von stdin

Listing 27: Kopieren (1) (copy1.c)

```
#include <stdio.h>

/* Beenden mit CTRL+D */
int main(void)
{
        int c;
        c = getchar();
        while (c != EOF) {
                putchar(c);
                c = getchar();
        }
        return 0;
}
```

Listing 28: Kopieren (2) (copy2.c)

```
#include <stdio.h>

int main(void)
{
        int c;
        while ((c = getchar()) != EOF) {
                /* Wehe wenn die Klammern
                 * fehlen
                 * while (c = getchar() != EOF) */
                putchar(c);
        }
        return 0;
}
```

Listing 29: Zeichen zählen (1) (wordcount1.c)

```c
#include <stdio.h>
#include <time.h>

/* wordcount < /etc/passwd */

int main(void)
{
        clock_t cputime, g = CLOCKS_PER_SEC;
        long counter=0;
        float rcputime;
        cputime = clock();
        while (getchar() != EOF)
                ++counter;
        printf("CLOCKS_PER_SEC: %li\n", g);
        printf("Zahl der Zeichen: %li\n", counter);
        rcputime = (float) (clock()-cputime)/g;
        printf("CPU-time: %f\n", rcputime);
        return 0;
}
```

Listing 30: Zeichen zählen (2) (wordcount2.c)

```c
#include <stdio.h>

int main(void)
{
        int counter;
        for (counter = 0; getchar() != EOF; ++counter);
        printf("Zahl der Zeichen: %i\n", counter);
        return 0;
}
```

Listing 31: Zeichen zählen (3) (wordcount3.c)

```
#include <stdio.h>

int main(void)
{
int wcounter, lcounter = 0, c;
for (wcounter = 0; (c = getchar()) != EOF; ++wcounter)
        {
        if (c == '\n')
                ++lcounter;
        }
printf("Zahl der Zeichen: %i\n", wcounter);
printf("Zahl der Zeilen: %i\n", lcounter);
return 0;
}
```

6.2 Speicherklassen

Die Typenerklärungen können noch näher spezifiziert werden, nämlich mit einer Speicherklasse, die etwas über ihre Lebensdauer und Bindung an andere Programmteile aussagt, und Attributen, die (systemabhängig) für Optimierungen Hilfestellungen geben.
Speicherklassen sind `auto, register, static, extern, typedef`, Attribute `const` und `volatile`.

Automatische Variablen

Die in Blöcken oder Funktionen deklarierten Objekte heißen automatisch und verlieren ihre Gültigkeit beim Verlassen. Sie werden mit jedem Aufruf neu erzeugt bzw. initialisiert. Die Speicherklasse `auto` ist die Voreinstellung für interne Objekte.

Register

Mit `register` werden externe oder interne automatische Variablen erklärt, die (wenn möglich) in schnellen Registern gehalten werden sollen. Der Adreßoperator `&` kann auf so erklärte Objekte nicht angewandt werden.

Externe Variablen

Wird ein Objekt in einer Quelldatei vor seiner Definition, die in einer anderen Quelldatei erfolgt, benutzt, so muß es zusätzlich mit `extern` vereinbart

werden.

Neue Datentypen

werden mit `typedef` vereinbart.

Attribute

Mit `const` werden Objekte vereinbart, die initialisiert, aber nicht verändert werden dürfen. Sie können also nicht Ziel einer Zuweisung sein. Mit `volatile` können Objekte vor einer (systemabhängigen) Optimierung geschützt werden. Es gibt z. Zt. keine implementierungsunabhängige Semantik dieses Attributes.

Adreßraum eines Programms	
code	Programm im Maschinencode
data	globale Daten
stack	lokale Variablen
heap	dynamische Variablen

6.3 Zeichen- und Zeichenfolgen

6.4 Datentyp char

Um Zeichen in einem Rechner darzustellen, wird jedem Zeichen eine Zahl zugeordnet. Mit dem ASCII-Code (7 bit Code) wurden 128 Zeichen wie folgt standardisiert.

Kontrollzeichen

0	^@			1	^A				
2	^B			3	^C	abbruch			
4	^D			5	^E				
6	^F			7	^G	bell	'\a'		
8	^H	backspace	'\b'	9	^I	htab	'\t'		
10	^J	newline	'\n'	11	^K	vtab	'\v'		
12	^L	formfeed	'\f'	13	^M	Return	'\r'		
14	^N			15	^O				
16	^P			17	^Q	xon			
18	^R			19	^S	xoff			
20	^T			21	^U				
22	^V			23	^W				
24	^X			25	^Y				
26	^Z	eof		27	^[				
28	^\			29	^]				
30	^^			31	^_				
127	^?	del							

Zeichen

32		33	!	34	"	35	#	36	$	37	%
38	|	39	’	40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/	48	0	49	1
50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=
62	>	63	?	64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G	72	H	73	I
74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U
86	V	87	W	88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_	96	‘	97	a
98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m
110	n	111	o	112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w	120	x	121	y
122	z	123	{	124	|	125	}	126	~		

Weitere Zeichen können über die Zahlen 128 bis 255 vereinbart werden (8 bit Code). Nicht alle dieser Zeichen können auf dem Bildschirm dargestellt werden. Variablen für ein Zeichen können über `char`, `signed char` bzw. `unsigned char` vereinbart werden. Die Schreibweise ist *’z’*, wobei *z* ein Zeichen repräsentiert. Es müssen `\, ?, ’, "` durch den `\` maskiert werden.

Character			
Variablentyp	Bytelänge	Minimum	Maximum
`char`	1	0	127
`signed char`	1	-128	127
`unsigned char`	1	0	255

Beispiel:
’ä’ entspricht:
als `char` -28 als `unsigned char` 228 als `signed char` -28

Umlaute					
ä	228	-28	ö	246	-10
ü	252	-4	Ä	196	-60
Ö	214	-42	Ü	220	-36
ß	223	-33			

Mit der Headerdatei `ctype.h` stehen folgende Funktionen zur Verfügung, die bei postivem Test eine Zahl > 0 zurückgeben.

Testfunktionen aus `ctype.h`	
`int isalnum(int char)`	Buchstaben und Zahlen
`int isalpha(int char)`	Buchstaben
`int isascii(int char)`	ASCII
`int iscntrl(int char)`	Kontrollzeichen
`int isdigit(int char)`	Ziffer
`int isgraph(int char)`	druckbar und kein blank
`int islower(int char)`	Kleinbuchstabe
`int isprint(int char)`	druckbar
`int ispunct(int char)`	weder alphanumerisch noch Blank
`int isspace(int char)`	Blank
`int isupper(int char)`	Großbuchstabe
`int isxdigit(int char`	Hexziffer

Umwandlungsfunktionen aus `ctype.h`		
`int`	`tolower(int char)`	wandelt in Kleinbuchstaben um
`int`	`toupper(int char)`	wandelt in Großbuchstaben um

6.5 Zeichenketten

Konstante Zeichenketten werden mit `"` begrenzt, intern werden sie als Vektoren dargestellt, die mit dem Zeichen `'\0'` beendet werden. Auf die einzelnen Zeichen einer Zeichenkette kann man somit über die Feldelemente zugreifen. Zeichenketten werden mit dem Format `%s` ausgegeben.

char *zkette*[10]; vereinbart eine Zeichenkette mit Namen *zkette*. Auf einzelne Zeichen kann mit *zkette[0]*, *zkette[1]*, ..., *zkette[10]* zugegriffen werden.

int strcmp(const char *s1, const char *s2) lexik. Vergleich: 0, wenn $s1 = s2$, > 0, wenn $s1 > s2$, < 0 wenn $s1 < s2$.

char *gets(char *var) liest die nächste Zeile in den Vektor `var` ein, dabei wird das Zeilenendezeichen durch das Endezeichen der Zeichenkette ersetzt. Bei einem Fehler wird der Pointer `NULL` zurück gegeben.

int puts(const char *var) schreibt die Zeichenkette `var` mit einem Zeilenendezeichen. Falls ein Fehler auftritt wird `EOF` zurück gegeben.

Listing 32: Beispiel (ampel.c)

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

int main(void)
{
char sfarbe[5];
int cf = 3;
enum farbe { rot, gruen, gelb };
printf("Eingabe einer Farbe: ");
gets(sfarbe);
puts(sfarbe);
if (!strcmp(sfarbe, "rot"))
        cf = 0;
else if (!strcmp(sfarbe, "gruen"))
        cf = 1;
else if (!strcmp(sfarbe, "gelb"))
        cf = 2;
switch (cf) {
case 0:
        printf("Halt\n");
        break;
case 1:
        printf("Start\n");
        break;
case 2:
        printf("Aufpassen\n");
        break;
default:
        printf("Sie stehen vor keiner Ampel\n");
}
        return 0;
}
```