

Vorlesungsmitschrift

# PROGRAMMIERUNG 1

Mitschrift von

**Falk-Jonatan Strube**

Vorlesung von

**Prof. Dr.-Ing. Arnold Beck**

5. Mai 2017

# INHALTSVERZEICHNIS

<b>1</b>	<b>Einführung</b>	<b>5</b>
1.1	Algorithmus . . . . .	5
1.2	Programmablaufplan (PAP) . . . . .	6
1.3	Struktogramm . . . . .	6
1.4	Quelltext in C . . . . .	6
<b>2</b>	<b>gcc</b>	<b>8</b>
<b>3</b>	<b>Grundlagen von C</b>	<b>9</b>
3.1	Eingebaute Datentypen . . . . .	9
3.2	Operationen . . . . .	10
3.3	Ausdrücke . . . . .	10
3.3.1	Assoziativität . . . . .	10
3.4	Anweisungen . . . . .	11
3.4.1	Ausdrucksanweisung . . . . .	12
3.4.2	Alternativanweisung . . . . .	12
3.4.3	Leeranweisung . . . . .	13
3.4.4	Iteration . . . . .	13
3.5	Zusammenfassendes Beispiel . . . . .	17
3.6	Zeichenketten . . . . .	18
3.7	Funktionen . . . . .	18
3.7.1	Gültigkeit . . . . .	19
3.8	Header-File . . . . .	20
3.9	Pointer . . . . .	20
3.10	Rekursion . . . . .	22
3.11	Benutzerdefinierte Datentypen . . . . .	23
3.11.1	Struct . . . . .	23
3.11.2	Union . . . . .	25
3.11.3	typedef . . . . .	26
3.11.4	enum . . . . .	26
3.12	Memory-Allocation . . . . .	27
<b>4</b>	<b>Dateiarbeit in C</b>	<b>29</b>
<b>5</b>	<b>Funktionspointer</b>	<b>32</b>
<b>6</b>	<b>Graphische Oberflächen</b>	<b>35</b>
6.1	libforms . . . . .	35
6.2	GTK . . . . .	35
6.3	CGI . . . . .	35
<b>7</b>	<b>Preprozessor</b>	<b>36</b>
7.1	Umwandlung von 3-Zeichenfolgen . . . . .	36
7.2	Zeilenverlängerung . . . . .	36
7.3	include . . . . .	36
7.4	Symboldefinitionen . . . . .	36



7.5	Parameterbehaftete Macros . . . . .	37
7.6	Bedingte Übersetzung . . . . .	39
<b>8</b>	<b>Funktionen mit variabler Argumentliste</b>	<b>40</b>
<b>9</b>	<b>CGI</b>	<b>43</b>



# HINWEISE

Zugelassene Hilfsmittel Klausur: Spickzettel A-4 Blatt, doppelseitig (, man-page c++.com)

VarArg nicht Prüfungsinhalt, da kein Praktikum dazu

Beleg abzugeben bis spätestens: 28.02. per Mail an [beck@informatik.htw-dresden.de](mailto:beck@informatik.htw-dresden.de) (Mail-Text: Studiengruppe + s-Nummer, Anhang: Quelldateien, Readme oder link zu Datei(en) in publichtml)



# 1 EINFÜHRUNG

Bilde Durchschnitt aus folgender Notenübersicht:

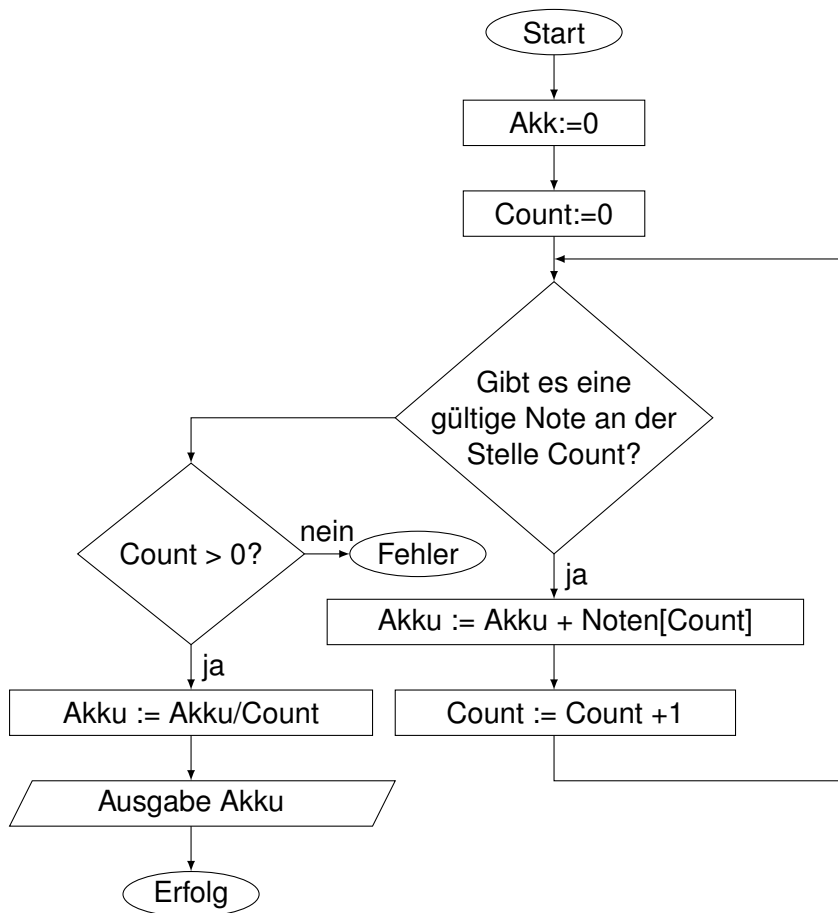
Index	Note
0	3
1	4
2	1
3	3
4	3
5	5
6	3
7	4
8	0
9	-

## 1.1 ALGORITHMUS

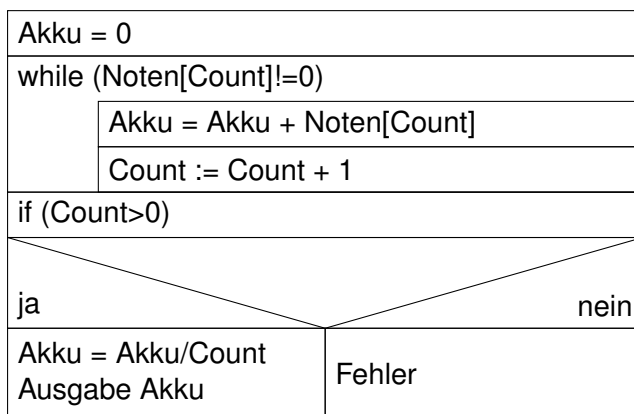
1. Lösche Akku  $\rightarrow$  2.
2. Lösche Counter  $\rightarrow$  3.
3. Gibt es eine Zahl an Stelle Count?
  - Ja:  $\rightarrow$  4.
  - Nein:  $\rightarrow$  6.
4. Addiere markierte Zahl zu Akku  $\rightarrow$  5.
5. Addiere 1 zu Counter  $\rightarrow$  3.
6. Dividiere Wert in Akku durch Wert in Counter und speichere Akku  $\rightarrow$  7.
7. Ergebnis: Ausgabe des Akku  $\rightarrow$  ENDE



## 1.2 PROGRAMMABLAUFPLAN (PAP)



## 1.3 STRUKTOGRAMM / NASSI-SHNEIDERMAN-DIAGRAMM



## 1.4 QUELLTEXT IN C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int Noten []={5,2,3,4,5,5,2,3,4,5,0}; //38/10
5

```



```

6 int main(){
7     int Akku=0, Count=0;
8     while (Noten[Count]!=0){
9         Akku = Akku+Noten[Count];
10        Count = Count+1;
11    }
12    if(Count>0){
13        Akku = Akku/Count;
14        printf("Durchschnitt: %d\n",Akku);
15    } else
16        printf("Fehler - Division durch 0\n");
17    return 0;
18 }

```

Compilieren durch:

```
1 gcc SOURCE.c -o DESTINATION
```

Ergebnis:

„3“ ... aber:  $38/10 = 3,8$ . Integer im Source-Code  $\Rightarrow$  abgerundet

Lösung:

```

1 // Ergebnis mit Runden (innerhalb der if(Count>0)-Klammer)
2 Akku = Akku*10/Count;
3 printf("Durchschnitt: %d.%d\n",Akku/10, Akku%10);

```

Alternativ:

```

1 // anstatt int Akku=0, Count=0;
2 double Akku=0;
3 int Count =0;

```



## 2 GCC

gcc Ablauf für eine „hello.c“ Datei.

1. Pre-Prozessor (hello.c → hello.e ⇒ gcc -E hello.c > hello.e)  
Alle Zeile im Quelltext mit # werden hier interpretiert.
2. Compiler (hello.e → hello.o ⇒ gcc -c hello.c)
3. Linker (hello.o → a.out / hello.exe | gcc hello.c → a.out ⇒ gcc -o hello hello.c [oder auch gcc hello.c -o hello])  
Bindet Objekt-Datei (xxx.o) mit Librarys zusammen.





# 3 GRUNDLAGEN VON C

- Großbuchstaben, Kleinbuchstaben, Ziffern, Sonderzeichen
- Ersatzsymbolfolgen (brauchen besondere Compileranweisung)
- white spaces:
  - Leerzeichen
  - Zeilenende(\n)
  - Tabulator(\t)
  - vertikaler Tab(\v)
  - Seitenvorschub(\f)
  - Bell(\a)
  - Backspace(\b)
  - Carriage return(\r)
- Schlüsselwörter:
  - auto, default, float, long, sizeof, union, break, do, for, register, static, ...

## 3.1 EINGEBAUTE DATENTYPEN

NATÜRLICHE ZAHLEN	Bytes	Min	Max
unsigned char	1	0	255
unsigned short int	2	0	65535
unsigned int	4	0	4294967295
unsigned long int	4 / 8	0	4294967295
unsigned long long int	8	0	1.8...*10 <sup>19</sup>

GANZE ZAHLEN	Bytes	Min	Max
signed char	1	-128	127
signed short int	2	-32768	32767
signed int	4	-2147483648	2147483647
signed long int	4 / 8	-21474836480	2147483647
signed long long int	8	-9.2*E18	9.2*E18

GEBROCHENE ZAHLEN	Bytes	VZ	Exp.	Mant	Wertebereich
float	4	1	8	23	1.2E-38 3.4E+38
double	8	1	11	52	2.3E-308 1.7E+308
Long double	10	1			3.4E-4932 1.1E+4932

Was grau markiert ist, kann weggelassen auch werden ⇒ „unsigned int i;“ = „unsigned i;“



### Bsp.:

```
1 unsigned int i; // Variablen-Definition
2 i = 12; // Wertzuweisung
3
4 printf("Wert von i: %d - Adresse von i: %p\n", i, &i);
5 // Hinweis:
6 // %d - Dezimalwert,
7 // %p - Adresswert,
8 // &i - Adresse von Variable
```

Erstellung einer Variablen (int i): UNINITIALISIERTE VARIABLE / VARIABLEN-DEFINITION  
Wertbelegung einer Variable während Definition einer Variablen (int i=0): INITIALISIERUNG  
Wertbelegung zu späterem Zeitpunkt (i=2): WERTZUWEISUNG

## 3.2 OPERATIONEN

Geordnet nach Priorität:

	Abarbeitung
() [] -> .	li nach re
! - (...) cast	
++ - * sizeof	re nach li
* / %	li nach re
+ -	li nach re
« »	li nach re
< <= > >=	li nach re
== !=	li nach re
&	li nach re
^	li nach re
	li nach re
&&	li nach re
	li nach re
?:	li nach re
= *= /= %= -= =	re nach li
^=  = <<= >>=	re nach li
,	

## 3.3 AUSDRÜCKE

Programmiersprachliche Konstruktion zur Berechnung von Werten.

### 3.3.1 ASSOZIATIVITÄT

(siehe Operatoren: Gewichtung der Operatoren von oben nach unten)

Unäre Operatoren (bspw. - (negativ-Zeichen), ++ (Inkrementierung) oder Klammern(cast))

Binäre Operatoren (bspw. +, - (Rechenzeichen), <= usw.)



```

1 int i;
2 long d;
3
4 i=(int)d; // cast: Typwandlung
5
6 i++; // Postfixoperator (wird im Rahmen eines groesseren
7     Ausdrucks als letztes ausgefuehrt:)
8 i=1;
9 j=6;
10 k=j+i++; // k=7, i=2
11 ++i; // Praefixoperator (wird im Rahmen eines groesseren
12     Ausdrucks als erstes ausgefuehrt:)
13 i=1;
14 j=6;
15 k=j+ ++i; // k=8, i=2
16
17 // Vorsicht! negativ-Bsp, wie ++ nicht zu verwenden ist:
18 i=2;
19 printf("%d\n", i++ + ++i);  \\ i= 6
20 printf("%d\n", i);  \\ i=4

```

Bei Division mit ganzen Zahlen wird der Rest abgeschnitten (nicht gerundet)!

Kurzschlussverfahren von Aneinanderkettung von Bedingungen (  $i < 0 \ || \ i < 6$  )  $\Rightarrow$  wenn die erste Prüfung wahr ist, wird der Test weiterer Bedingungen abgebrochen (bei  $\&\&$  wenn das erste falsch ist).

$\&\&$  im Vergleich zu  $\&$  ( $\&$  ist eine Bit-weise Operation):

$01101100 \ \& \ 00001111 = 00001100$  bzw.

$01101100 \ | \ 11110000 = 11111100$

Andere Zeichen:

$\wedge = \text{XOR}$

$\sim = \text{Bit-weise Negation}$

$\ll = \text{shift (nach links)}$  (bsp.  $i=4$ ;  $i = i \ll 2$ ;  $\Rightarrow i$  wird 16:

$00000100 \ll 2 \Rightarrow 00010000$

Achtung: bei negativen Zahlen (also Typ signed) bleibt bei Shift an der ersten Stelle das entsprechende Vorzeichenbit.

Bsp. für Abarbeitungsreihenfolge der Operatoren:

$i* = 3+1 \ // \ i*(3+1)$

### 3.4 ANWEISUNGEN

- Berechnungen
- Alternative
- Iteration
- Sequenz



### 3.4.1 AUSDRUCKSANWEISUNG (EXPRESSIONSTATEMENT)

Eine Ausdrucksanweisung besteht aus einem Ausdruck gefolgt von einem Semikolon:

```
1 <expr_stmt>:: <expr> ';' .
```

Bsp.:

```
1 printf("%d\n", i);
```

Zu Ausdrucksanweisungen gehören:

- Berechnungen
- Aufrufe von Funktionen

**Block** Konstruktion, die Anweisungen kapselt – nach außen einzelne Anweisungen enthält

- Vereinbarungen
- Anweisungen

```
1 <block>:: '{' { <statement> } '}' .
```

Bsp.:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char vbuf [128]; // Vereinbarung
5
6 int main() {
7     int i; // Vereinbarung
8     double x; // Vereinbarung
9     fgets(vbuf, 128, stdin); // Anweisungen ...
10    x=atof(vbuf);
11    i = 1;
12    x=x*10+i;
13    printf("x: %lf\n",x);
14 }
```

### 3.4.2 ALTERNATIVANWEISUNG (IF-STATEMENT)

```
1 <if_stmt>:: 'if' '(' <condition> ')'  
2     <statement>  
3     ['else' <statement>] .
```

Bsp.:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char vbuf [128];
5
6 int main() {
```



```

7  double x;
8  fgets(vbuf, 128, stdin);
9  x=atof(vbuf);
10 printf("x: %lf\n",x);
11 if (x>1) printf("Groesser als 1\n");
12 else printf("Kleiner als 1\n"); // optional
13 puts("Hier geht es weiter");
14 // " " Strings (Zeichenketten), einzelnes Zeichen: '*'
15 }

```

### 3.4.3 LEERANWEISUNG

```
1 <empty_stmt>:: ';'

```

### 3.4.4 ITERATION (SCHLEIFE/LOOP)

**abweisende Schleife (kopfgesteuert) while-Schleife**

```
1 <while_statement>:: 'while' '(' <condition> ')' <statement> .

```

Beispiel:  $e^x$

e hoch x =  $1 + x/1! + x*x/2! + x*x*x/3! \dots$

1. Summand:  $x^0 = 1$
2. Summand:  $x^1/1! = x$
3. Summand:  $(x^1/1!) * x/2 = x^2/2!$
4. Summand:  $(x^2/2!) + x/3 = x^3/3!$

Vereinfachung der Rechnung (für den Rechner)  $\Rightarrow$  Nutzung des vorhergehenden Summanden.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char vbuf[128];
5
6 int main() {
7     int i=1;
8     double x, y=1.0, summand = 1.0;
9     printf("Eingabe von x: ");
10    fgets(vbuf, 128, stdin);
11    x=atof(vbuf);
12    while (summand>0,00005){
13        summand = summand *x/i;
14        y += summand;
15        printf("Summand %d: %lf\n", i, summand);
16        i++;
17    }
18    printf("e^%lf: %lf\n", x, y);
19    return 0;
20 }

```



## Nicht abweisende Schleife (fußgesteuert) do-while-Schleife

```
1 <do-stmnt>:: 'do' <statement> 'while' '(' <condition> ')' ';' .
```

Bsp.:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char vbuf[128];
5
6 int main() {
7     int i=1;
8     double x, y=1.0, summand = 1.0;
9     printf("Eingabe von x: ");
10    fgets(vbuf, 128, stdin);
11    x=atof(vbuf);
12    do{
13        summand = summand *x/i;
14        y += summand;
15        printf("Summand %d: %lf\n", i, summand);
16        i++;
17    } while (summand>0,00005);
18    printf("e~%lf: %lf\n", x, y);
19    return 0;
20 }
```

## for-Schleife

```
1 <for-stmnt>:: 'for' '(' <expr>;' <expr>;' <expr> ')' <statement>
.
```

Bsp.:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char vbuf[128];
5
6 int main() {
7     int i=1;
8     double x, y=1.0, summand = 1.0;
9     printf("Eingabe von x: ");
10    fgets(vbuf, 128, stdin);
11    x=atof(vbuf);
12    for (i=1;    // Schleifeninitialisierung
13        summand > 0.0005;    // Abbruchbedingung / Condition
14        i++){ // Iterationsausdruck
15        summand = summand *x/i;
16        y += summand;
17        printf("Summand %d: %lf\n", i, summand);
18    }
19    printf("e~%lf: %lf\n", x, y);
```



```

20 return 0;
21 }

```

Alternativ-Bsp der for-Schleife mit Komma-Operator:

```

1 int main() {
2   int i=1;
3   double x, y, summand;
4   printf("Eingabe von x: ");
5   fgets(vbuf, 128, stdin);
6   x=atof(vbuf);
7   for (i=1, y=1.0, summand=1.0;
8       summand > 0.0005;
9       summand*=x/i, y+=summand,
10      printf("Summand %d: %lf\n", i, summand), i++){
11  }
12  printf("e~%lf: %lf\n", x, y);
13  return 0;
14 }

```

**Verlassen der Schleife break**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char vbuf [128];
5
6 int main() {
7   int i=1;
8   double x, y=1.0, summand = 1.0;
9   printf("Eingabe von x: ");
10  fgets(vbuf, 128, stdin);
11  x=atof(vbuf);
12  while (1){
13    summand = summand *x/i;
14    y += summand;
15    printf("Summand %d: %lf\n", i, summand);
16    if (summand<0,00005) break;
17    i++;
18  }
19  printf("e~%lf: %lf\n", x, y);
20  return 0;
21 }

```

BREAK bezieht sich auf die (von innen nach außen) nächste zu findende Schleife. Also auf die Schleife, in deren STATEMENT sie vorkommt.

**Neuberechnung der Bedingung continue**

Verlässt den Schleifenkörper (der eingebettete Anweisung) und prüft die Bedingung erneut.

```

1 #include <stdio.h>
2 #include <stdlib.h>

```



```

3
4 char vbuf[128];
5
6 int main() {
7     int i=1;
8     double x, y=1.0, summand = 1.0;
9     printf("Eingabe von x: ");
10    fgets(vbuf, 128, stdin);
11    x=atof(vbuf);
12    while (summand>0,00005){
13        summand = summand *x/i;
14        y += summand;
15        i++;
16        if (summand > 0.00005) continue;
17        printf("Summand %d: %lf\n", i-1, summand);
18    }
19    printf("e^%lf: %lf\n", x, y);
20    return 0;
21 }

```

Wenn Summand größer als 0.00005 ist, startet er die Schleife neu. Die printf() wird erst ausgeführt, wenn er kleiner ist (also das letzte mal).

### Fallunterscheidung switch-Anweisung

```

1 switch (i){ // i ist ganzzahliger Ausdruck
2     case 1:
3         ... break;
4     case 2:
5         ... break;
6     default:
7         ...
8 }

```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char buf[128];
5
6 int main(){
7     int wota;
8     printf("Wochentag (1...7): ");
9     fgets(buf, 128, stdin); wota=atoi(buf);
10    switch (wota){
11        case 1: puts("Montag");      break;
12        case 2: puts("Dienstag");    break;
13        case 3: puts("Mittwoch");    break;
14        case 4: puts("Donnerstag");  break;
15        case 5: puts("Freitag");     break;
16        case 6: puts("Samstag");     break;
17        case 7: puts("Sonntag");     break;
18

```





```

19     default: puts("Die Woch hat nur 7 Tage!");
20 }
21 return 0;
22 }

```

### 3.5 ZUSAMMENFASSENDEN BEISPIEL

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char buf [128];
5
6 int main(){
7     int result=0;
8     char operator=0;
9     int value;
10    while (operator!=toupper('q')){
11        printf("Eingabe Operator: ");
12        fgets(buf, 128, stdin);
13        operator = buf[0];
14        printf("Eingabe Zahl: ");
15        fgets(buf, 128, stdin);
16        value = atoi(buf);
17        switch (operator) {
18            case '+': // Erinnerung: kein "+" - nur '+' für einzelne
                Zeichenketten
19                result += value;
20                break;
21            case '-':
22                result -= value;
23                break;
24            case '*':
25                result *= value;
26                break;
27            case '/':
28                if (value) // bzw. value!=0 - aber !=0 kann in C
                weggelassen werden
29                result /= value;
30            else
31                puts("Division durch 0 ist nicht erlaubt.");
32            break;
33            case '%':
34                if (value) // bzw. value!=0 - aber !=0 kann in C
                weggelassen werden
35                result %= value;
36            else
37                puts("Division durch 0 ist nicht erlaubt.");
38            break;
39            case 'q':
40                break;

```



```

41     default:
42         printf("unerlaubte Operation %c\n", operator);
43     }
44     printf("result: %d\n", result);
45 }
46 }

```

## 3.6 ZEICHENKETTEN

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 char buf [128];
6
7 int main(){
8     printf("Eingabe Zeichenkette: ");
9     fgets(buf, 128, stdin);
10    printf("Len von Str: %d\n", strlen(buf));
11    buf[strlen(buf)-1]=0;
12    puts(buf);
13    while (buf[i]!=0)
14        printf("%c", buf[i++]);
15    printf("\n");
16    return 0;
17 }

```

Bei der Eingabe „Max“ wird bei puts() sowohl die Eingabe der Zeichenkette, als auch die Eingabe der Eingabetaste (neue Zeile) ausgegeben.

M	a	x	\n	∅			
---	---	---	----	---	--	--	--

mit ∅: binäre, terminierende Null ( 0000 0000 )

Hinweis: der Buffer muss immer noch Platz für „\n“ und „∅“ haben, d.h. man hat in einem Buffer der Größe von 128 nur Platz für 126 zeichen.

mit BUF[STRLEN(BUF)-1]=0; wird die Eingabetaste „\n“ raus gelöscht.

Daraus ergibt sich eine Verbesserung für den Taschenrechner:

```

1 ...
2     printf("Eingabe Operator /Operand: ");
3     fgets(buf, 128, stdin);
4     operator = buf[0];
5     value = atoi(buf+1); // Buffer ab der Stelle 0+1: 1
6 ...

```

## 3.7 FUNKTIONEN

= Unterprogramme, zur Wiederholung von Codepassagen und zur besseren Strukturierung.

```

1 <function> :: <func_head><funcbody>.

```



```

2 <func_head> :: [<return_type>] name '('<param_lst> |void|')'.
3 <param_list> :: [<type_name> name {','<type_name> name}]
4 <func_body> :: <block>

```

Bsp.:

```

(func_head)    int    main()
               (return_type)
               {
(func_body)    return 0;
               }

```

Wenn kein return\_type gewählt wurde, dann default: INT.

Wenn kein return\_type gebraucht wird, gibt man VOID an.

```

1 long fakult (int x) { // int x: Parameter
2     long f=1;
3     int i;
4     for (i=1; i<=x; i++){
5         f*=i;
6     }
7     return f;
8 }
9
10 char vBuf [128];
11
12 int main(){
13     double x,y;
14     printf("Eingabe x: ");
15     fgets(buf,128,stdin); x= atof(vBuf);
16     y =fakult(x);
17     printf("y: %ld \n", y);
18     return 0;
19 }

```

### 3.7.1 GÜLTIGKEIT

- Bereich im C-Quelltext, an dem ein Bezeichner sichtbar ist.
- Lebensdauer: Zeit von Erzeugung bis zur Vernichtung

```

1 static int count; // default wert 0
2 count++; // behält jedes Mal ihren Wert, im Gegensatz zu int count
!

```

#### Speicherklassen:

AUTO (automatische Variable): wird vom Stack erzeugt (Kellerspeicher)

lokale Variablen

EXTERN: Variable, die in einem anderen Kontext vereinbart ist

STATIC: leben bis zum Programmende, global-statische Variablen werden nicht exportiert, immer initialisiert, default 0

REGISTER: Variablen werden nach Möglichkeit in ein Prozessorregister gelegt (schnell)

VOLATILE: Variablen werden immer im Speicher abgelegt



```

1 long fakult(int x); //Funktionsdeklaration
2     // (prototyp)

```

### 3.8 HEADER-FILE

enthält die Funktionsköpfe

```

1 #include "fe.h" // wie bspw. stdio.h kann die eigene Datei in
   anderen
2     // Quelltexten eingebunden werden

```

### 3.9 POINTER

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int i=99, *pi=&i;
6     // &: Adressoperator
7     printf("i: %d &i: %p \n", i, &i);
8     // folgendes gibt nicht 99, sondern die Adresse von i (und die
   eigene) aus.
9     // erst über *pi wird der Wert des Pointers ausgegeben (->
   Dereferenzieren)
10    printf("pi: %lx &pi: %p \n *pi: %d", pi, &pi, *pi);
11    // Äquivalent dazu: (Arrays und Pointer sind in C sehr eng
   miteinander verwandt)
12    printf("pi: %lx &pi: %p \n pi[0]: %d", pi, &pi, pi[0]);
13
14    return 0;
15 }

```

**Beispiel** Beispiel anhand Eingangsbeispiel der Vorlesung für das Berechnen eines Durchschnitts.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int Noten []={5,2,3,4,5,5,2,3,4,5,0}; //38/10
5
6 // double CalcMean(int Noten[]){
7 // folgendes präziser, da eigentlich kein Array, sondern bloß ein
   Pointer übergeben wird,
8 // der Pointer ist eine neue Variable, die auf die Adresse des
   ersten Elements
9 // des Arrays zeigt (gilt für Parameter (wie hier) und return-Werte
   )

```



```

10 // int n für die Weitergabe der Array-Länge
11 double CalcMean(int* Noten, int n){
12     double Akku=0.0;
13     int Count=0;
14     printf("Len von Noten in Funktion: %d\n", sizeof Noten); // = 4
15     // = 4, weil nur die Adresse des ersten Elements! Kein
        Information über
16     // die Länge des Arrays durch die Übergabe in die Funktion!!!
17     while (Count < n){
18         // Obwohl Noten nur Pointer ist, können wir trotzdem Noten[
            Count] verwenden
19         Akku = Akku+Noten[Count];
20         // Alternativ: (Pointer-Dereferenzierung und Pointer-Arithmetik
            )
21         // * hier keine Operation, sondern der Dereferenzierungs-Stern
22         // Akku = Akku + *(Noten+Count);
23         // auch Alternativ: Pointer-Variable wird weiter gestellt (
            diese ist nicht global)
24         // Akku = Akku + *Noten++;
25         // Folgende Zeile würde das globale Array ändern!!! (s.u.)
26         // Noten[Count] = 0;
27         Count = Count+1;
28     }
29     if(Count>0)
30         // Ergebnis mit Runden
31         Akku = Akku/Count;
32     return Akku;
33 }
34
35 int main(){
36     double mean = 0.0;
37
38     // Die tatsächliche Länge des Arrays
39     printf("Len von Noten in Main: %d\n", sizeof Noten); // = 44
40
41     // ACHTUNG: Wenn wir das Array an die Funktion übergeben, wird
        nicht das Array selbst
42     // übergeben, sondern die Adresse des Arrays (bzw. des ersten
        Elements)! Wird also
43     // innerhalb der Funktion das Array geändert, wird auch das
        globale Array geändert
44     mean = CalcMean(Noten, sizeof(Noten)/sizeof(int));
45     printf("Durchschnitt: %lf\n", mean);
46     // sizeof(...): Operator, der die gröÙe eines Elements bestimmt.
47     // Durch das teilen durch sizeof(int) erhält man die Anzahl der
        Elemente
48     for (i=0; i<sizeof(Noten)/sizeof(int); i++)
49         printf("%d ", Noten[i]);
50     puts("");
51     return 0;
52 }

```



## 3.10 REKURSION

Beispiel der Fakultät mit Rekursion:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 long fakultr(int x){
5     long f;
6     if (x>1)
7         f = x*fakultr(x-1);
8     else
9         f=1;
10    return f;
11 }
12
13 // Bsp. Abarbeitung:
14 // - u: 120
15 //   u/10: 12 >0
16 // - u: 12
17 //   u/10: 1 >0
18 // - u: 1
19 //   u/10: 0 = 0
20 // Rekursiv werden dann die ASCII-Codes ausgegeben (ASCII-Wert von
21 //   0 + Zahl)
22 void printu(unsigned int u){
23     if (u/10>0)
24         printu(u/10);
25     putchar(u%10+'0');
26 }
27
28 int main(){
29     long f;
30     int i = 5;
31     f = fakultr(i);
32     printu((unsigned int) f);
33     puts("");
34 }
```

- Rechtsrekursion (erst etwas rechnen, dann in die Rekursion gehen → fakultr)
- Linksrekursion (erst Rekursiv aufrufen, dann etwas ausführen → printu)

Eine Links- oder Rechtsrekursion lässt sich auch iterativ darstellen.

- Zentralrekursion

Eine Zentralrekursion lässt sich nicht iterativ darstellen.

Problem Rekursion: Es lässt sich nicht vorhersehen, wie viel Speicher benötigt wird. Da ist die Schleife leichter überschaubar.



## 3.11 BENUTZERDEFINIERTER DATENTYPEN

### 3.11.1 STRUCT

Struktur: alle Elemente liegen hintereinander (nicht zwangsläufig unmittelbar hintereinander) im Speicher

Alle kompilieren:

gcc \*.c (dafür braucht es die Header Datei st1.h, die alle Funktionen enthält [außer main]) ⇒

alle Dateien werden in eine kompiliert

st1.h:

```
1 typedef unsigned int uint; // wie unten beispiel für synonym
   definition
2
3 typedef // Wenn typedef davor steht, muss unterhalb ein Typ
   angegeben werden
4 struct{ //tStudent{ wenn typedef angegeben, kann der Typname davor
   weggelassen werden.
5     char name[28+1];
6     int matrNr:15; // hier bspw. uint, anstatt unsigned int (wenn
   benötigt)
7     int belnote:3; // :x gibt die Bitlänge des Integer-Wertes ein
8     // (falls Ressourcen knapp) Bezeichnung: Bitfelder
9     float klNote;
10    // Struktur innerhalb der Struktur:
11    struct {
12        int jj,stGang, stGr;
13    } stGr;
14 } tStud; // <- hier der Typ, synonym zu tStud. Jetzt kann überall
   anstatt
15    // "struct tStudent" nur "tStud" geschrieben werden
16
17 void putStud(tStud s); // hier kann nun die Verkürzung angewandt
   werden
18    //-> aber auch in Dokumenten, in denen dieses included wird.
19 void putStudp(tStud s);
20 void getStud1(tStud * pStud);
21 tStud getStud2();
```

st1.c:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "st1.h"
4
5 struct tStudent stud ={"Hans Huckebeil", 12121, 0,
6     0.0,{2015,41,1}};
7 tStud ProjGr[5] ={"Hans Huckebeil", 12121},
8     {"Peter Lehmann", 12122},
9     {"Anna Bauer", 12123}};
10
11 int main(){
12     ProjGr[3]=getStud2();
```



```

12 ProjGr[4]=getStud2();
13
14 int i;
15 for (i=0; i<5; i++){
16     putStud(ProjGr[i]);
17 }
18 for (i=0; i<5; i++){
19     puts(ProjGr[i].name);
20 }
21 // grÖÙe des Studenten zum Prüfen von Bitfeldern
22 printf("Size Student: %d \n",sizeof(tStud));
23 // ohne Bitfelder: 44
24 // mit Bitfelder: 36
25 // grÖÙe einer Struktur immer mit SIZEOF ermitteln! Nicht selbst
    ausrechnen
26 // Compiler füllt Lücken ein (siehe 2. Übung)
27 printf("%s %d/%d/%d \n",stud.name, stud.stGr.jj, stud.stGr.stGang
    , stud.stGr.stgr);
28
29 //putStud(stud);
30 //getStud1(&stud);
31 //putStudp(&stud);
32 return 1;
33 }

```

st1f.c:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "st1.h"
5
6 static char buf[128];
7
8 void putStd(struct tStudent s){
9     printf("%-28s %u \n", s.name,s.matrNr);
10 }
11
12 void putStudp(tStud *s){
13     //printf("%-28s %u \n", (*s).name,(*s).matrNr); alternativ dazu:
14     printf("%-28s %u \n", s->name,s->matrNr);
15 }
16
17 void getStud1(tStud * pStud){
18     printf("Eingabe Name: ");
19     fgets(vbuf,128,stdin);
20     buf[strlen(buf)-1]=0; // wegschmeiÙen des "\n" der Eingabe
21     // Arrays können nicht zugewiesen werden, deshalb muss
22     // über Funktion strcpy() Byte für Byte eingetragen werden:
23     strcpy(pStud->name,buf);
24     printf("Eingabe Matrikelnr.: ");
25     fgets(vbuf,128,stdin);

```





```

26     pStud->matrNr=atoi(buf);
27 }
28
29 tStud getStud2(){
30     tStud s;
31     printf("Eingabe Name: ");
32     fgets(vbuf,128,stdin);
33     buf[strlen(buf)-1]=0;
34     strcpy(s.name,buf);
35     printf("Eingabe Matrikelnr.: ");
36     fgets(vbuf,128,stdin);
37     s.matrNr=atoi(buf);
38     return s;
39 }

```

### 3.11.2 UNION

Alle Elemente eines Union-Datentyps liegen übereinander und liegen auf einer Adresse.

tu1.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef union{
5     unsigned int i;
6     unsigned char arr[5];
7 } tbsp;
8
9 tbsp x;
10
11 // Argumente der main: Nutzen durch Programmzeilen Parameter
12 // argc: Anzahl der Parameter
13 // argv[i]: Der entsprechende Parameter
14 // Aufruf: ./tu1.c Max
15 // argc = 2, argv[0]=./tu1.c , argv[1]=Max
16 int main(int argc, char* argv[]){
17     int i;
18     for (i=0; i<argc; i++){
19         puts (argv[i]);
20     }
21
22     if (argc==2){
23         x.i = strtod(argv[1]); // strtod alternativ zu atoi
24         for (i=0; i<4; i++){
25             printf("%02x ",x.arr[i]); // so liegt die Zahl im Speicher
26         }
27         puts("");
28     }
29     return 0;
30 }

```



Sinnvoll, wenn Unterstrukturen gleiche und auch ungleiche Eigenschaften haben. Beispielsweise Diplom- und Bachelor-Studenten.

### 3.11.3 TYPEDEF

Für Details s.o.

Zusätzlicher Trick für den Umgang mit Arrays, bzw. sizeof() und typedef.

```
1 typedef int tIntArr[6];
2
3 void dispArr(tIntArr x){
4     int i;
5     printf("length (Byte): %d length (Elements): %d\n",sizeof(IntArr
6         ),
7         sizeof(IntArr)/sizeof(int));
8     // da Referenz auf typedef und nicht auf x, funktioniert sizeof
9     ohne Probleme
10    for (i=0; i<sizeof(IntArr)/sizeof(int); i++)
11        printf("%d ",x[i]);
12    puts("");
13 }
14
15 int main (){
16     tIntArr arr={2,4,6,8,10,12}
17     dispArr(arr);
18     return 0;
19 }
```

### 3.11.4 ENUM

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 enum tWoTa{
5     Montag=1, // bei Montag=1, fängt's bei 1 an zu zählen (anstatt 0)
6     Dienstag,
7     Mittwoch,
8     Donnerstag,
9     Freitag,
10    Samstag=Freitag+1+0x10, // verändert das Wochenende
11    Sonntag};
12
13 enum tWoTa w=1;
14
15 int main(){
16     printf("%d %x \n",Montag, Montag); // gibt standarmäßig 0 aus;
17     Sonntag wäre 6 usw.
18     printf("%d %x \n",Dienstag, Dienstag);
19     printf("%d %x \n",Mittwoch, Mittwoch);
20     printf("%d %x \n",Donnerstag, Donnerstag);
21     printf("%d %x \n",Freitag, Freitag);
```



```

21 printf("%d %x \n",Samstag, Samstag);
22 printf("%d %x \n",Sonntag, Sonntag);
23 w=99;
24 printf("%d %x \n",w, w);
25 return 0;
26 }

```

## 3.12 MEMORY-ALLOCATION

Aus vorhergehendem Beispiel:

st1.c:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "st1.h"
4
5 int anz = 0;
6 char vbuf[128];
7 char weiter = 'y'; // Ja
8
9 tStud s;
10
11 int main(){
12     int i;
13     tStud *ps = NULL, *psx;
14     while (weiter == 'y'){
15         s = getStud2();
16         if(ps==NULL){
17             ps=malloc(sizeof(tStud)); // Malloc Speicher aus dem heap
18             if (ps) {puts("malloc hat nicht geklappt."); exit(-1);}
19         } else {
20             psx=realloc(ps, (anz+1)*sizeof(tStud));
21             // realloc guckt nach, ob dahinter noch genug Speicher frei
                ist und
22             // fügt das neue alloc hinten dran. Wenn nicht, dann wird ein
                neuer Speicherbereich
23             // gefunden, der groß genug ist für beide ist.
24             if (psx){
25                 ps=psx;
26             } else { puts("realloc hat nicht geklappt."); exit(-1);}
27         }
28
29         *(ps+anz) = s;
30         anz++;
31         for (i=0; i<anz; i++) {
32             putStudp(ps+i);
33             printf("i: %d, ps+i: %p, sizeof: %d %02x\n",i, ps+i, sizeof(
                tStud),sizeof(tStud));
34         }
35         printf("next (y/n): ");
36         fgets(vbuf,128,stdin);

```



```

37     weiter=vbuf[0];
38 }
39 // am Ende des Programms oder wenn die Daten nicht mehr gebraucht
    werden
40 // muss der Speicher wieder freigegeben werden
41 free (ps);
42 // free (psx); darf nicht freigegeben werden, da er nur ein
    Zeiger auf den ps ist.
43 return 0;
44 }

```

#### Verwendung von malloc/realloc:

Speicher nach Bedarf aus dem heap.

malloc hat als Ausgabe void\* (generischer Pointer) [siehe "man malloc"]. Dieser ist nicht dereferenzierbar und zuweisungskompatibel zu jedem getypten Pointer. Man kann mit ihm ebenfalls nicht rechnen (keine Arithmetik).

malloc: Speicher für Variable frei machen

realloc: freigemachten Speicher erweitern

free: Speicher wieder freigeben

Es kann bspw. auch Speicher in getStud2 (z.B. für den Namen, also \*name anstatt name mit entsprechendem alloc in der Fkt.) allociert werden, um Platz zu sparen (oder dynamisch mehr freizugeben). Dann müsste der Speicher entsprechend in einer Schleife auch wieder freigegeben werden.



## 4 DATEIARBEIT IN C

```
1 FILE * pf;  
2 pf=fopen("myFile.txt","rt"); // ("Dateiname inkl. Pfad", "Modus")
```

b/t	Texdatei ([t]) / Binärdatei(b)
r	Zum Lesen öffnen
w	Zum Schreiben (überschreiben) öffnen, ggf. erzeugen
a	Zum Schreiben am Dateiende öffnen, ggf. erzeugen
r+	Zum Lesen und Schreiben öffnen (ändern)
w+	Zum Lesen und Schreiben öffnen (überschreiben), ggf. erzeugen
a+	Zum Anfügen, Lesen, Erzeugen öffnen, ggf. erzeugen

**Als Binärdatei speichern** aus tStud Beispiel(st1.c) (jeweils an den entsprechenden Stellen eingefügt):

```
1 #include "stdio.h"  
2 ...  
3 FILE *pf;  
4  
5 int main(){  
6     int i;  
7     tStud* ps = NULL, *psx;  
8     pf = fopen("Studs.bin","rb");  
9     if(pf){  
10        // Dateigröße ermitteln:  
11        fseek(pf,0,SEEK_END);  
12        anz=ftell(pf)/sizeof(tStud);  
13        rewind(pf);  
14  
15        // Daten lesen  
16        for (i=0; i<anz; i++){  
17            if(ps==NULL){  
18                ps=malloc(sizeof(tStud));  
19                if (ps==NULL){puts("malloc hat nicht geklappt"); exit(-1);}  
20                *ps=readStud(pf);  
21            } else {  
22                psx=realloc(ps,(anz+1)*sizeof(tStud));  
23                if (psx){  
24                    ps=psx;  
25                    *(ps+i)=readStud(pf);  
26                } else { puts("realloc hat nicht geklappt."); exit(-1);}  
27            }  
28        }  
29        fclose(pf);
```



```

30 }
31 ...
32 // Daten speichern
33 pf=fopen("Studs.bin","wb");
34 if(pf==NULL){puts("fopen (write) hat nicht geklappt");exit(-1);
35 for (i=0; i<anz; i++){
36     writeStud(pf,ps+i);
37 }
38 fclose(pf);
39 ...

```

### st1io.h

```

1 #include <stdio.h>
2 #include "st1f.h"
3 #include "st1io.h"
4
5 tStud readStud(FILE* f){
6     tStud s;
7     fread(&s, sizeof(tStud),1,f);
8     return s;
9 }
10
11 void write Stud(File* f, tStud* pStud){
12     fwrite(pStud, sizeof(tStud),1,f);
13 }

```

**ACHTUNG:** Arbeiten mit derartigen Binärdateien ist sehr Plattform-Abhängig (angucken der Binärdatei unter linux mit "hexdump -C ...")

**Als Textdatei speichern** (alternativ und ergänzend zum obigen Beispiel)

### st1io.h

```

1 int getAnz(FILE* pf);
2 tStud readStud(FILE* pf);
3 void writeStud(FILE* pf, tStud* ps);

```

### st1io.c

```

1 #include "st1f.c"
2
3 // Zeilen zählen und durch 4 teilen
4 int getAnz(FILE *pf){
5     char buf[128];
6     int n=0;
7     while (fgets(buf, 128, pf)) n++;\\
8     fseek(pf, 0, SEEK_SET);
9     return n/4;
10 }
11
12 tStud readStud(FILE* pf){
13     tStud s={};
14     char buf[128];
15     if(fgets(buf, 128, pf)){

```



```

16     buf[strlen(buf)-1]=0;
17     s.name = malloc(strlen(buf+1));
18     if (s.name) strcpy(s.name,buf);
19     else fprintf(stderr, "malloc fauld in readStud\n");
20     fscanf(pf, "%d\n%d\n%f\n", &s.matrNr, &s.belNote, &s.klNote);
21 }
22 return s;
23 }
24
25 void writeStud(FILE *pf, tStud* ps){
26     fprintf(pf, "%s\n%d\n%d\n%f\n", ps->name, ...
27 }

```

### st1.c

```

1 ...
2 // Einlesen aus Datei
3 pf=fopen("Studs.txt", "rt");
4 if (pf){
5     anz=getAnz(pf);
6 ...
7 // Schreiben
8 pf=fopen("Studs.txt", "wt");

```

**Als CSV-Datei speichern** Ähnlich wie bei .txt, bloß trennt man die Datensätze durch Semicolon und nicht durch neue Zeile.

Sinnvolle Funktion zur Zerlegung der Datensätze:

```

1 strtok(buf, "; \n");

```



# 5 FUNKTIONSPONTER

Bisher wurde typedef für Variablen verwendet:

```
1 typedef struct{
2     int sk1;
3     int sk2;
4 } tStruct;
5 tStruct s={1,9999};
```

Nun kommt der Funktionspointer dazu:

```
1 typedef void f(void);
2 f* pf; // Das ist der Funktionspointer
```

oder auch:

```
1 typedef void (*tpf)(void);
2 tpf pf; // Das ist auch ein Funktionspointer
```

Anwendungsbeispiel:

```
1 void fxyz(void){
2     puts("xyz");
3 }
4 pf = fxyz;
5
6 fxyz();
7 pf(); // diese beiden Funktiosaufrufe rufen die gleiche Funktion
      // auf - ein mal dirket
8     //und ein mal als Pointer
```

**Beispiel:**

```
1 #include <stdio.h>
2
3 typedef void (*tpf)(int i);
4 tpf pFunc;
5
6 void printDec(int i);
7 void printHex(int i);
8
9 typedef struct{
10     tpf func;
11     int x;
12 }tstruct;
13
14 tstruct tdec={printDec,55};
15 tstruct thex={printHex,55};
```





```

16
17 void printDec(int i){
18     printf("%d\n",i);
19 }
20
21 void printHex(int i){
22     printf("%x\n",i);
23 }
24
25 // alternativ: printx(void (*func)(int i), int z){...}
26 void printx(tpf func, int z){
27     func(z);
28 }
29 // Bezeichnung der Technik: Übergabe einer Callback Funktion
30 // Verwendung bei Programmierung mit graphischen Oberflächen
31
32 tpf choice(char c){
33     if (c=='h') return printHex;
34     else         return printDec;
35 }
36
37 int main(int argc, char*argv[]){
38     int x=9999;
39     tpf aFctns[]={printDec,printHex};
40     if (argc==2) x=atoi(argv[1]);
41
42     pFunc=printDec;
43     puts("Dezimal:");
44     pFunc(x);
45     pFunc=printHex;
46     puts("Hexadezimal:");
47     pFunc(x);
48
49     puts("Dezimal aus Array:");
50     aFctns[0](x);
51     puts("Hexadezimal aus Array:");
52     aFctns[1](x);
53
54     puts("Dezimal aus Funktionsparameter:");
55     printx(printDec,x);
56     puts("Hexadezimal aus Funktionsparameter:");
57     printx(printHex,x);
58
59     puts("aus Struct:");
60     tdec.func{tdec.x};
61     thex.func{thex.x};
62
63     puts("aus Returnwert:");
64     choice('h')(x);    // choice('h')()=printHex()
65     choice('d')(x);
66

```



```
67 return 0;  
68 }
```

Verwendungszweck beispielsweise im Compilerbau im Lexer oder ähnliches.



# 6 GRAPHISCHE OBERFLÄCHEN

## 6.1 LIBFORMS

FOLIE

## 6.2 GTK

FOLIE

## 6.3 CGI

FOLIE

Bsp. `ascii.cgi`



# 7 PREPROZESSOR

## 7.1 UMWANDLUNG VON 3-ZEICHENFOLGEN

(siehe Grundelemente): Bspw. ??> o.ä.  
Braucht extra Befehl für gcc.

## 7.2 ZEILENVERLÄNGERUNG

Falls eine Zeichenkette im Editor mit Zeilenumbruch getrennt werden soll (bspw. ein langer String). Kann auch mitten im C-Code sein.  
Das Backslash muss dafür das letzte Zeichen auf der Quelltextzeile sein.

```
1 #include <stdio.h>
2
3 int ma\
4 in(){
5     puts{Spass\
6     beim Programmieren\
7     in c");
8     return 0;
9 }
```

## 7.3 INCLUDE

Der Preprozessor ersetzt die Include-Zeile mit dem Quelltext des einzubindenden Elements.

```
1 #include <...> // für Systemheaderfiles (zu finden unter usr/
   include)
2 #include "... " // für Applikationsheaderfiles (eigene)
```

Das Ergebnis des Procompilers kann mit GCC -E PROGR.C > PROGR.E kompiliert werden.  
Mehrfaches Einbinden u.ä. kann unterbunden werden durch:

```
1 #ifndef _H_DEBUG_
2 #define _H_DEBUG_
3
4 #include <stdio.h>
5
6 #endif
```

## 7.4 SYMBOLDEFINITIONEN

Vorteile:



- Quelltext wird lesbarer (durch gute zweckgebundene Bezeichnung)
- Symboldefinition kann im gesamten Quelltext mit einem Mal geändert werden.

```

1 // Allgemein:
2 #define SYMBOL Tokensequenz // Symbol wird üblicher Weise groß
   geschrieben,
3                               // muss aber nicht (nur zur besseren
                               Wiedererkennung).
4
5 // Beispiel:
6 #define N 10
7 ...
8 int inArray[N]; // Preprozessor ersetzt N mit 10
9
10 #define LEN 30 + 1
11 // KLAMMERN SETZTEN! ->
12 // #define LEN (30 + 1)
13 ...
14 char name[LEN*3]; // Achtung: LEN wird vorm Ausrechnen ersetzt.
15                   // Also: LEN*3 entspricht 30+1*3 und nicht (30+1)
                   *3
16 strcpy(name, "Max");
17 puts(name);
18 printf("Len name: %d, Len content: %d\n", sizeof name, strlen(name)
   );
19
20 #define NAME1 Klaus
21 ...
22 puts(NAME1);

```

Darauf ist zu achten:

- Tokensequenz (bei Zahlen) am besten Klammern.
- kein Semikolon!
- define-Konstruktion muss auf einer Zeile stehen.

## 7.5 PARAMETERBEHAFTETE MACROS

```

1 #define SYMBOL(<parameterlist>) Ersatztokenfolge
2 // Wichtig: Runde Klammer muss unmittelbar hinter SYMBOL stehen
3 #define SYMBOL(x) "str1" #x "str2"
4 // Bewirkt Verkettung der Strings mit dem Parameter
5 #define SYMBOL(x,y) x##y
6 // Ein neues Token entsteht im C-Quelltext aus x und y

```

Beispiel:

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <math.h>

```



```

4 #define LEN (30+1)
5 #define PYTHAGORAS(a,b) sqrt(a*a + b*b)
6 // Klammern wieder wichtig:
7 // #define PYTHAGORAS(a,b) sqrt((a)*(a) + (b)*(b))
8 #define STR1(x) "Max " #x " Moritz"
9 // Raute vor dem Parameter bewirkt das Zusammenfassen der
   umschlossenen Strings.
10 // (wird zu "Max " "und" " Moritz")
11 #define STR2(x,y) x##y
12 #define MYPUTS puts
13
14 int main(){
15     double x=3;
16     double y=4;
17     printf("c:  %lf\n",PYTHAGORAS(x,y));
18     printf("c:  %lf\n",PYTHAGORAS(x+1,y+1));
19     // wenn nicht geklammert: PYTHAGORAS(x+1,y+1) = sqrt(x+1*x+1+y+1*
   y+1)
20     // also sqrt(2x+1+2y+1) -> Falsch!
21     printf("c:  %lf\n",PYTHAGORAS(x++,y++));
22     // auf keinen Fall machen, da es schlecht überschaubar wird, wann
   inkrementiert wird!
23     printf("c:  %lf\n",PYTHAGORAS(x,funktion(y)));
24     // auch gefährlich, weil der Macro mit funktion(y)*funktion(y)
   ersetzt wird
25     // und damit die funktion (unerwünscht) doppelt ausgeführt wird
26
27     char name [LEN*3];
28     strcpy(name, STR1(und));
29     puts(name);
30
31     STR2(MY,PUTS)("Test"); // ergibt MYPUTS("Test"), was dann mit
   puts() ersetzt wird.
32
33     return 0;
34 }

```

Erinnerung:

bei #INCLUDE MATH.H muss mit GCC PROGR.C -LM kompiliert werden!

Darauf ist zu achten:

- Parameter im Macro klammern
- keine Seiteneffekte programmieren (Inkrementierung, Funktionsaufruf usw.)

Liste von vordefinierten Symbolen zu finden durch GCC -E -DM - </DEV/NULL.

Bspw.:

```

1 __FILE__ // Quelltext-Dateiname
2 __LINE__ // Zeilennummer, in der das Symbol verwendet wurde
3 __DATE__ // Datum, zu dem das Programm kompiliert wurde
4 __TIME__ // Zeit, zu der das Programm kompliert wurde
5 __STDC__ // C-Version

```



## 7.6 BEDINGTE ÜBERSETZUNG

Soll ein plattformübergreifendes Programm geschrieben werden (mit einem Quelltext), gibt es Tücken (bspw. 32 oder 64 bit). Dann können bestimmte Fälle mit einer bedingten Übersetzung abgefangen werden.

```
1 #if <const-expr>
2 #if defined <symbol>
3 #ifdef <symbol>
4 #if !defined <symbol>
5 #ifndef <symbol>
6 #else
7 #elif <const-expr>
8 #endif
```

```
1 #define MYPUTS puts
2 ...
3 #ifdef MYPUTS
4     MYPUTS("Text");
5 #else
6     printf("Text2");
7 #endif
```

Symbole können mit der Commandlineoption -D auch definiert werden.  
GCC `PROGR.C -D MYPUTS=PUTS` (entspricht der Zeile `#DEFINE MYPUTS PUTS`).

Gutes Anwendungsbeispiel für bedingte Übersetzung:

```
1 #ifdef DEBUG
2     printf("Debuginformation");
3 #endif
```

Gibt Debug-Information nur bei `GCC PROGR.C -DDEBUG` aus.



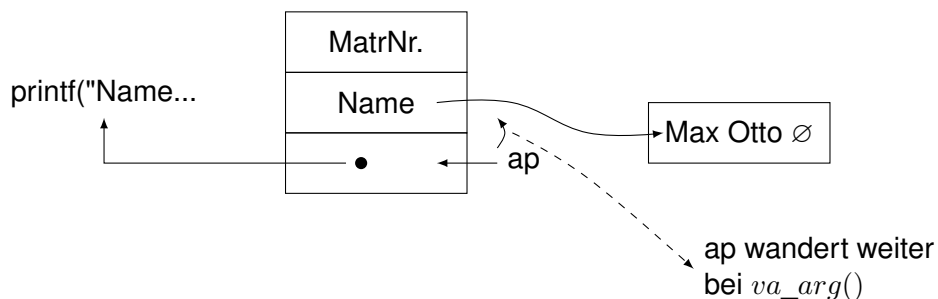
## 8 FUNKTIONEN MIT VARIABLER ARGUMENTLISTE

vgl.: printf() mit beliebig vielen Argumenten abhängig von % d usw. im ersten String.

1. wenigstens ein fester Parameter
2. dann folgt, ...  
(Es können also weitere Parameter folgen. Typ und Anzahl der Parameter ist unbekannt.)

Macros zum Umgang mit variabler Argumentliste:

```
1 #include <stdarg.h> // Voraussetzung für variable Argumentlisten
2
3 va_start(ap, la);
4 // ap: Argument Pointer ( va_list ap; )
5 // la: Last Argument (letzter Parameter mit Typ und Namen vor ...)
6 printf("Name: %s, MatrNr: %d \n", Stud.name, Stud.MatrNr);
```



```
7 x=va_arg(ap, type);
8 // x: Zielvariable für den Wert des Arguments
9 // (muss vom Typ des tatsächlichen Parameters sein)
10 // ap wird um sizeof(type) erhöht
11 // dieser Wert wird x zugewiesen
```

```
12 va_end(ap);
```

**Beispiel:** myprintf.c

```
1 //#include <stdio.h> // Kann auskommentiert werden, da
   // entsprechende Funktionen
   // in recout.h nachprogrammiert wurden.
2
3 #include <stdarg.h>
4 #include <recout.h> // Andere my...-Funktionen in Arg.tgz
5 // als Systemheader-file, mit Befehlen aus
   // librecout.c rein kopiert
6 #define xprintf(x) myprintf x
7
```





```

8 void myprintf(const char* fmt, ...){ // fmt bspw. ("Programm: %s\n
   ", argv[0])
9   va_list ap;
10  char *p;
11  char *pstr;
12  char cval;
13  int ival;
14  double dval;
15
16  va_start(ap,fmt);
17
18  for ( p=fmt; // p auf Anfang des Formatsteuerstrings
19        *p;
20        p++){
21    if (*p!='%'){
22      myputc(*p);
23    } else {
24      switch(++p){
25        case 's':
26          pstr=va_arg(ap, char*);
27          myputs(pstr);
28          break;
29        case 'd':
30          ival=va_arg(ap, int);
31          myputd(ival);
32          break;
33        case 'x':
34          ival=va_arg(ap, int);
35          myputx(ival);
36          break;
37        case 'd':
38          dval=va_arg(ap, double);
39          myputdouble(dval);
40          break;
41        case 'c':
42          cval=va_arg(ap, int); // char wird zu int geändert
43          myputc(cval);
44          break;
45        case 'p':
46          pstr=va_arg(ap, char*);
47          myputx((long)pstr); // muss gecastet werden
48          break;
49        default:
50          myputc(*p);
51          break;
52      }
53    }
54  }
55 }
56
57 int main(int argc, char* argv[]){

```



```

58  int i;
59  if (argc>1)
60      i = myatoi(argv[1]);
61  else
62      i = -1;
63
64  double d=13.37;
65
66  myprintf("Programm: %s\n int Value: %d\n", argv[0], argc);
67  myprintf("i: %d, %x, DoubleVal: %f, Char: %c,
68           Adresse argv[0]: %p",i,i,d, argv[0][0], argv[0]);
69
70  xprintf( ("Test xprintf: %s\n", argv[0]) );
71  return 0;
72 }

```

Zum Kompilieren: Anlinken der Bibliothek: GCC MYPRINTF.C -LRECOUT (librecout.a wird zu lrecout). Include übernimmt nur Prototypen – die kompilierte Library muss so manuell übergeben werden (stdio-Library wird beim include automatisch angehängt).

In librecout.c:

```

1 // gcc -c librecout.c
2 // ar rvs librecout.a librecout.c
3 // sudo cp librecout.a /usr/local/lib/librecout.a
4 // sudo cp recout.h /usr/local/include

```



## 9 CGI

<http://141.56.64.101/cgi-bin/cam.cgi>

CGI Programm mit dem ein Text (über Telefon-Baustein) an ein entferntes anderes Telefon-Baustein gesendet wird (von dem das dann interpretiert wird)

SMSLV.HTML in CGI-BIN

```
1 <html>
2 <head>
3   <title> sms cgi 27.01.2016 </title>
4 </head>
5 <body>
6   <h3>snd sms from rasperry</h3>
7   <form name "smsForm" action="smslv.cgi" METHODE="POST" id="
8     smsform">
9     <table cellpadding="4" cellspacing="0">
10      <tr>
11        <td style="width:100px"><label>to PhoneNr</label></td>
12        <td style="width:600px"><input type="text" name="smsToPhone">
13      </td></tr>
14      <tr>
15        <td style="width:100px"><label>Text</label></td>
16        <td style="width:600px"><textarea rows="2" cols="80"
17          name="smsText" id="smsForm"></textarea></td>
18      </tr>
19      <tr>
20        <td><input type="submit" value="Senden"></td>
21      </tr>
22    </table>
23  </form>
24 </body>
25 </html>
```

SMSLV.C in CGI-BIN

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 FILE *f, *fsms;
6 char param[256];
7 char phone[20];
8 char text[256];
9 char buf[256];
10 char *p, *q;
```



```

11
12 #ifndef DEBUG
13 #define xprintf(x) printf x
14 #else
15 #define xprintf(x)
16 #endif
17
18 int readText(char *p, char *q){
19     int j;
20     for (j=0; *p!=0; p++, j++)
21     switch (*p){
22         case '+': q[j]=' '; break;
23         case '%': p++; q[j]=(char)strtol(p,&p,16); break;
24         default: q[j]=*p; break;
25     }
26     q[j]=0;
27     return j-1;
28 }
29
30 int main(){
31
32     puts("Content-Type: text/html\n\n");
33     fgets(param,256,stdin); // html-Formulardaten landen in stdin
34     if (strlen(param,>2){
35         xprintf(("Query: %s<br>", param))
36         q=phone;
37         // Telefonnummer holen:
38         p=strstr(param,"smsToPhone")+strlen("smsToPhone"); // strstr:
           suche im String
39         // +strlen(): nicht auf den Anfang des Textes zeigend,
40         // sondern auf die erste Zahl der Nummer (also nach dem Text)
41         while(*p && *p!='&'){
42             *q++ = *p++
43         }
44         xprintf(("Phone: %s<br>", phone));
45         // Text holen:
46         p=strstr(param,"smsText")+strlen("smsText");
47         readText(p,text);
48         xprintf(("Text: %s<br>", text);
49
50         fsms=fopen("/var/spool/gnokii/sms.txt","wt");
51         if (fsms){
52             fprintf("%s\n%s", phone, text);
53             fclose(fsms);
54         } else{
55             printf("Could not open sms.txt");
56         }
57     }
58     f=fopen("smslv.html","rt");
59     if (f) {
60         while(fgets(buf,256,f))

```



```
61     puts(buf);
62 } else{
63     puts("Could not open smslv.html");
64 }
65 return 0;
66 }
```

