Programmierung I

Vorlesungsskript

Mitschrift von Falk Jonatan Strube

Vorlesung von Prof. Dr.-Ing. Beck

25. Januar 2016

Inhaltsverzeichnis

Mitschrift von
Falk Jonatan Strube

Hinweise

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

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.

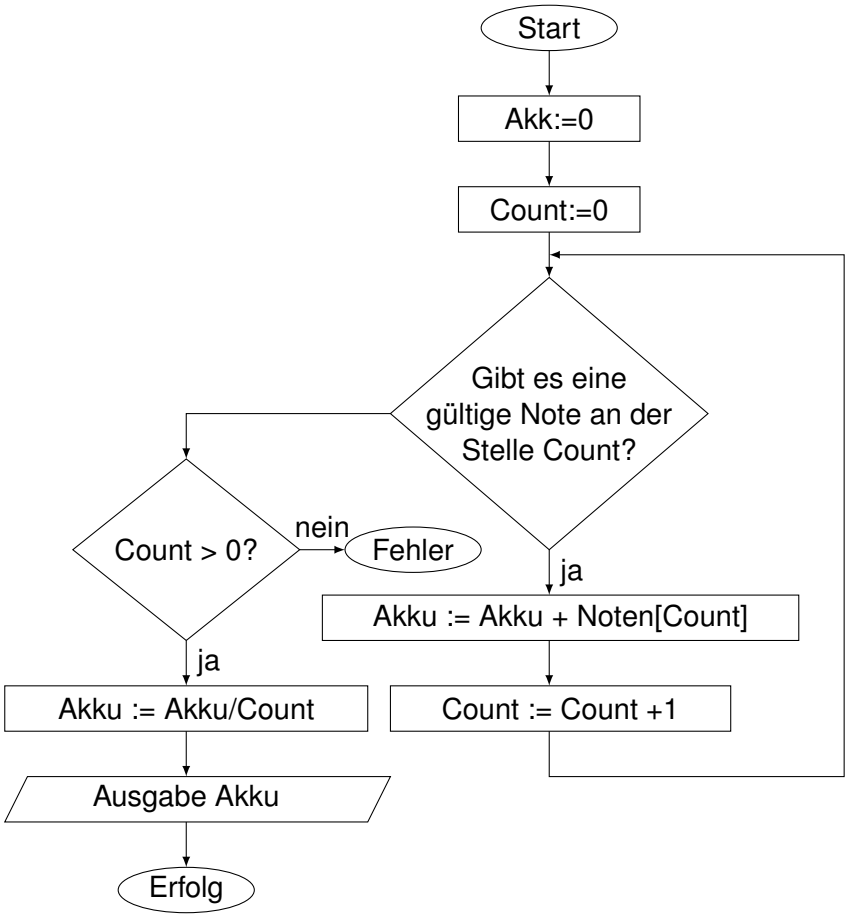
5.) Addiere 1 zu Counter $\rightarrow$ 3.

6.) Dividiere Wert in Akku durch Wert in Counter und speichere Akku $\rightarrow$ 7.

7.) Ergebnis: Ausgabe des Akku $\rightarrow$ ENDE

Mitschrift von
Falk Jonatan Strube

1.2 Programmablaufplan (PAP)



1.3 Struktogramm / Nassi-Shneiderman-Diagramm

Akku = 0	
while (Noten[Count]!=0)	
Akku = Akku + Noten[Count]	
Count := Count + 1	
if (Count>0)	
ja	nein
Akku = Akku/Count Ausgabe Akku	Fehler

1.4 Quelltext in C

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

int Noten []={5,2,3,4,5,5,2,3,4,5,0}; //38/10

int main(){
  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 -0 DESTINATION
```

Ergebnis:
„10“ . . . aber: 38/10 = 3,8. Integer im Source-Code ⇒ 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)
Jede 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

FOLIE „Grundlagen von C“

3.1 Eingebaute Datentypen

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

Mitschrift von
Falk Jonatan Strube

Bsp.:

```
unsigned int i; // Variablen-Definition
i = 12;    // Wertzuweisung

printf("Wert von i: %d - Adresse von i: %p\n", i, &i);
// Hinweis:
// %d - Dezimalwert,
// %p - Adresswert,
// &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 Ausdrücke

Programmiersprachliche Konstruktion zur Berechnung von Werten.

3.2.1 Assoziativität

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

Unäre Operatoren (bspw. $-$ (negativ-Zeichen), $++$ (Inkrementierung) oder Klammern(cast))
Binäre Operatoren (bspw. $+$, $-$ (Rechenzeichen), $<=$ usw.)

```
int i;
long d;

i=(int)d; // cast: Typwandlung

i++; // Postfixoperator (wird im Rahmen eines groesseren
      Ausdrucks als letztes ausgefuehrt:)
i=1;
j=6;
k=j+i++; // k=7, i=2
++i; // Praefixoperator (wird im Rahmen eines groesseren
      Ausdrucks als erstes ausgefuehrt:)
i=1;
j=6;
k=j+ ++i; // k=8, i=2

// Vorsicht! negativ-Bsp, wie ++ nicht zu verwenden ist:
i=2;
printf("%d\n", i++ + ++i);   \\ i= 6
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:
$\wedge$ = XOR
$\sim$ = Bit-weise Negation
<< = shift (nach links) (bsp. i=4; i= i << 2; $\Rightarrow$ i wird 16:
00000100 << 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)

3.3 Anweisungen

- Berechnungen
- Alternative
- Iteration
- Sequenz

3.3.1 Ausdrucksanweisung (Expressionstatement)

Eine Ausdrucksanweisung besteht aus einem Ausdruck gefolgt von einem Semikolon:

```
<expr_stmnt>:: <expr> ';' .
```

Bsp.:

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

Zu Ausdrucksanweisungen gehören:

- Berechnungen
- Aufrufe von Funktionen

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

- Vereinbarungen
- Anweisungen

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

Bsp.:

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

char vbuf[128]; // Vereinbarung

int main() {
  int i;  // Vereinbarung
  double x; // Vereinbarung
  fgets(vbuf, 128, stdin);  // Anweisungen ...
  x=atof(vbuf);
  i = 1;
  x=x*10+i;
  printf("x: %lf\n",x);
}
```

3.3.2 Alternativanweisung (if-statement)

```
<if-stmnt>:: 'if' '(' <condition> ')'
        <statement>
      ['else' <statement>] .
```

Bsp.:

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

char vbuf[128];

int main() {
  double x;
  fgets(vbuf, 128, stdin);
  x=atof(vbuf);
  printf("x: %lf\n",x);
  if (x>1) printf("Groesser als 1\n");
  else printf("Kleiner als 1\n"); // optional
  puts("Hier geht es weiter");
  // " " Strings (Zeichenketten), einzelnes Zeichen: '*'
}
```

3.3.3 Leeranweisung

```
<empty_stmnt>:: ';'
```

3.3.4 Iteration (Schleife/Loop)

abweisende Schleife (kopfgesteuert) while-Schleife

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

Beispiel: e^x
e hoch x = 1 + x/1! + x*x /2! + x*x*x /3! ...

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.

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

char vbuf[128];

int main() {
  int i=1;
  double x, y=1.0, summand = 1.0;
  printf("Eingabe von x: ");
  fgets(vbuf, 128, stdin);
  x=atof(vbuf);
  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:

```c
int main() {
  int i=1;
  double x, y, summand;
  printf("Eingabe von x: ");
  fgets(vbuf, 128, stdin);
  x=atof(vbuf);
  for (i=1, y=1.0, summand=1.0;
  summand > 0.0005;
  summand*=x/i, y+=summand,
      printf("Summand %d: %lf\n", i, summand), i++){
  }
  printf("e^%lf: %lf\n", x, y);
  return 0;
}
```

Verlassen der Schleife break

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

char vbuf[128];

int main() {
  int i=1;
  double x, y=1.0, summand = 1.0;
  printf("Eingabe von x: ");
  fgets(vbuf, 128, stdin);
  x=atof(vbuf);
  while (1){
    summand = summand *x/i;
    y += summand;
    printf("Summand %d: %lf\n", i, summand);
    if (summand<0,00005) break;
    i++;
  }
  printf("e^%lf: %lf\n", x, y);
  return 0;
}
```

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.

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

char vbuf[128];

int main() {
  int i=1;
  double x, y=1.0, summand = 1.0;
  printf("Eingabe von x: ");
  fgets(vbuf, 128, stdin);
  x=atof(vbuf);
  while (summand>0,00005){
    summand = summand *x/i;
    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

```
switch (i){ // i ist ganzzahliger Ausdruck
  case 1:
    ... break;
  case 2:
    ... break;
  default:
    ...
}
```

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

char buf[128];

int main(){
  int wota;
  printf("Wochentag (1...7): ");
  fgets(buf, 128, stdin); wota=atoi(buf);
  switch (wota){
    case 1: puts("Montag");      break;
    case 2: puts("Dienstag");    break;
    case 3: puts("Mittwoch");    break;
    case 4: puts("Donnerstag"); break;
    case 5: puts("Freitag");     break;
    case 6: puts("Samstag");     break;
    case 7: puts("Sonntag");     break;

    default: puts("Die Woch hat nur 7 Tage!");
  }
  return 0;
}
```

3.4 Zusammenfassendes Beispiel

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

char buf [128];

int main(){
  int result=0;
  char operator=0;
  int value;
  while (operator!=toupper('q')){
    printf("Eingabe Operator: ");
    fgets(buf, 128, stdin);
    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  }
```

3.5 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:

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

3.6 Funktionen

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

```
<function> :: <func_head><funcbody>.
<func_head> :: [<return_type>] name '('<param_lst> void ')'.
<param_list> :: [<type_name> name {','<type_name> name}]
<func_body> :: <block>
```

Bsp.:

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

Wenn kein return_type gewählt wurde, dann default: *int*.
Wenn kein return_type gebraucht wird, gibt man *void* an.

```
long fakult (int x) { // int x: Parameter
  long f=1;
  int i;
  for (i=1; i<=x; i++){
    f*=i;
  }
  return f;
}

char vBuf[128];

int main(){
  double x,y;
  printf("Eingabe x: ");
  fgets(buf,128,stdin); x= atof(vBuf);
  y =fakult(x);
  printf("y: %ld \n", y);
  return 0;
}
```

3.6.1 Gültigkeit

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

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

Speicherklassen:
auto (automatische Variable): wird vom Stack erzeugt (Kellerspeicher)
lokale Varibalen
extern: Variable, die in einem anderen Kontext vereinbart ist
static: leben bis zum Programmende, global-statische Variabln werden nicht exportiert, immer initialiesiert, default 0
register: Variablen werden nach Möglichkeit in ein Prozessorregister gelegt (schnell)
volatile: Variabeln werden immer im Speicher abgelegt

```
long fakult(int x); //Funktionsdeklaration
          // (prototyp)
```

3.7 Header-File

enthält die Funktionsköpfe

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

3.8 Pointer

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

int main(){
  int i=99, *pi=&i;
  // &: Adressoperator
  printf("i: %d &i: %p \n", i, &i);
  // folgendes gibt nicht 99, sondern die Adresse von i (und die eigene) aus.
  // erst über *pi wird der Wert des Pointers ausgegeben (-> Dereferenzieren)
  printf("pi: %lx &pi: %p \n *pi: %d", pi, &pi, *pi);
  // Äquivalent dazu: (Arrays und Pointer sind in C sehr eng miteinander verwandt)
  printf("pi: %lx &pi: %p \n pi[0]: %d", pi, &pi, pi[0]);

  return 0;
}
```

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

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

int Noten []={5,2,3,4,5,5,2,3,4,5,0}; //38/10

// double CalcMean(int Noten[]){
// folgendes präziser, da eigentlich kein Array, sondern bloß ein Pointer übergeben wird,
// der Pointer ist eine neue Variable, die auf die Adresse des ersten Elements
// des Arrays zeigt (gilt für Parameter (wie hier) und return-Werte)
// int n für die Weitergabe der Array-Länge
double CalcMean(int* Noten, int n){
  double Akku=0.0;
  int Count=0;
  printf("Len von Noten in Funktion: %d\n", sizeof Noten);  // = 4
  // = 4, weil nur die Adresse des ersten Elements! Kein Information über
  // die Länge des Arrays durch die Übergabe in die Funktion!!!
  while (Count < n){
    // 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 größe 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  }
```

3.9 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 0 + Zahl)
21  void printu(unsigned int u){
22    if (u/10>0)
23      print(u/10);
```

```
24   putchar(u%10+'0');
25 }
26
27 int main(){
28   long f;
29   int i = 5;
30   f = fakultr(i);
31   printu((unsigned int) f);
32   puts("");
33 }
```

- 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.

3.10 Benutzerdefinierte Datentypen

3.10.1 Struct

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

Alle komplieren:
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 Verkürzug 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:

```c
#include <stdio.h>
#include <stlib.h>
#include "st1.h"

struct tStudent stud ={"Hans Huckebeil", 12121, 0, 0.0,{2015,41,1}};
tStud ProjGr[5] ={{"Hans Huckebeil", 12121},
          {"Peter Lehmann", 12122},
          {"Anna Bauer", 12123}};

int main(){
  ProjGr[3]=getStud2();
  ProjGr[4]=getStud2();

  int i;
  for (i=0; i<5; i++){
    putStud(ProjGr[i]);
  }
  for (i=0; i<5; i++){
    puts(ProjGr[i].name);
  }
  // größe des Studenten zum Prüfen von Bitfeldern
  printf("Size Student: %d \n",sizeof(tStud));
  // ohne Bitfelder: 44
  // mit Bitfelder: 36
  // größe einer Struktur immer mit SIZEOF ermitteln! Nicht selbst ausrechnen
  // Compiler füllt Lücken ein (siehe 2. Übung)
  printf("%s %d/%d/%d \n",stud.name, stud.stGr.jj , stud.stGr.stGang, stud.stGr.stgr);

  //putStud(stud);
  //getStud1(&stud);
  //putStudp(&stud);
  return 1;
}
```

st1f.c:

```c
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include "st1.h"

static char buf[128];

void putStd(struct tStudent s){
  printf("%-28s %u \n", s.name,s.matrNr);
}

void putStudp(tStud *s){
  //printf("%-28s %u \n", (*s).name,(*s).matrNr); alternativ dazu:
  printf("%-28s %u \n", s->name,s->matrNr);
}

void getStud1(tStud * pStud){
  printf("Eingabe Name: ");
  fgets(vbuf,128,stdin);
  buf[strlen(buf)-1]=0; // wegschmeißen des "\n" der Eingabe
  // Arrays können nicht zugewiesen werden, deshalb muss
  // über Funktion strcpy() Byte für Byte eingetragen werden:
  strcpy(pStud->name,buf);
  printf("Eingabe Matrikelnr.: ");
  fgets(vbuf,128,stdin);
  pStud->matrNr=atoi(buf);
}
```

```
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 }
```

3.10.2 Union

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

tu1.c

```
1  #include <stdio.h>
2  #include <stlib.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.

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

3.10.4 enum

```
1  #include <stdio.h>
2  #inculde <stlib.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; Sonntag wäre 6 usw.
17   printf("%d %x \n",Dienstag, Dienstag);
18   printf("%d %x \n",Mittwoch, Mittwoch);
19   printf("%d %x \n",Donnerstag, Donnerstag);
20   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 }
```

3.11 Memory-Allocation

Aus vorhergehendem Beispiel:
st1.c:

```
1  #include <stdio.h>
2  #include <stlib.h>
3  #include "st1.h"
4
5  int anz = 0;
6  char vbuf[128];
7  char weiter ='y'; // Ja
8
9  tStud s;
10
```

```c
int main(){
  int i;
  tStud *ps = NULL, *psx;
  while (weiter == 'y'){
    s = getStud2();
    if(ps==NULL){
      ps=malloc(sizeof(tStud)); // Malloc Speicher aus dem heap
      if (ps) {puts("malloc hat nicht geklappt."); exit(-1);}
    } else {
      psx=realloc(ps,(anz+1)*sizeof(tStud));
      // realloc guckt nach, ob dahinter noch genug Speicher frei ist und
      // fügt das neue alloc hinten dran. Wenn nicht, dann wird ein neuer Speicherbereich
      // gefunden, der groß genug ist für beide ist.
      if (psx){
        ps=psx;
      } else { puts("realloc hat nicht geklappt."); exit(-1);}
    }

    *(ps+anz) = s;
    anz++;
    for (i=0; i<anz; i++) {
      putStudp(ps+i);
      printf("i: %d, ps+i: %p, sizeof: %d %02x\n",i, ps+i, sizeof(tStud),sizeof(tStud));
    }
    printf("next (y/n): ");
    fgets(vbuf,128,stdin);
    weiter=vbuf[0];
  }
  // am Ende des Programms oder wenn die Daten nicht mehr gebraucht werden
  // muss der Speicher wieder freigegeben werden
  free (ps);
  // free (psx); darf nicht freigegeben werden, da er nur ein Zeiger auf den ps ist.
  return 0;
}
```

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

FOLIE

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

```c
#include "st1io.h"
...
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 Platform-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

```
#include "st1f.c"

// Zeilen zählen und durch 4 teilen
int getAnz(FILE *pf){
  char buf[128];
  int n=0;
  while (fgets(buf, 128, pf)) n++;\\
  fseek(pf, 0, SEEK_SET);
  return n/4;
}

tStud readStud(FILE* pf){
  tStud s={};
  char buf[128];
  if(fgets(buf, 128, pf)){
    buf[strlen(buf)-1]=0;
    s.name = malloc(strlen(buf+1));
    if (s.name) strcpy(s.name,buf);
    else fprintf(stderr, "malloc faild in readStud\n");
    fscanf(pf,"%d\n%d\n%f\n",&s.matrNr,&s.belNote,&s.klNote);
  }
  return s;
}

void writeStud(FILE *pf, tStud* ps){
  pfrintf(pf,"%s\n%d\n%d\n%f\n",ps->name, ...
}
```

st1.c

```
...
  // Einlesen aus Datei
  pf=fopen("Studs.txt", "rt");
  if (pf){
    anz=getAnz(pf);
...
  // Schreiben
  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:

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

5 Funktionspointer

Bisher wurde typedef für Variablen verwendet:

```
typedef struct{
  int sk1;
  int sk2;
} tStruct;
tStruct s={1,9999};
```

Nun kommt der Funktionspointer dazu:

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

oder auch:

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

Anwendungsbeispiel:

```c
void fxyz(void){
  puts("xyz");
}
pf = fxyz;

fxyz();
pf(); // diese beiden Funktiosaufrufe rufen die gleiche Funktion auf – ein mal dirket
      //und ein mal als Pointer
```

Beispiel:

```c
#include <stdio.h>

typedef void (*tpf)(int i);
tpf pFunc;

void printDec(int i);
void printHex(int i);

typedef struct{
  tpf func;
  int x;
}tstruct;

tstruct tdec={printDec,55};
tstruct thex={printHex,55};

void printDec(int i){
  printf("%d\n",i);
}

void printHex(int i){
  printf("%x\n",i);
}

// alternativ: printx(void (*func)(int i), int z){...}
void printx(tpf func, int z){
  func(z);
}
// Bezeichnung der Technik: Übergabe einer Callback Funktion
// Verwendung bei Programmierung mit graphischen Oberflöchen

tpf choice(char c){
  if (c=='h') return printHex;
  else        return printDec;
}

int main(int argc, char*argv[]){
  int x=9999;
  tpf aFctns[]={printDec,printHex};
  if (argc==2) x=atoi(argv[1]);

```

```
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-FOLIE). 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.

```c
#include <stdio.h>

int ma\
in(){
   puts{Spass\
   beim Programmieren\
   in c");
   return 0;
}
```

7.3 include

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

```c
#include <...>  // für Systemheaderfiles (zu finden unter usr/include)
#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:

```c
#ifdefn _H_DEBUG_
#define _H_DEBUG_

#include <stdio.h>

#endif
```

7.4 Symboldefinitionen

Vorteile:

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

```c
// Allgemein:
#define SYMBOL Tokensequenz    // Symbol wird üblicher Weise groß geschrieben,
                               // muss aber nicht (nur zur besseren Wiedererkennung).

// Beispiel:
#define N 10
...
int inArray[N]; // Preprozessor ersetzt N mit 10

#define LEN 30 + 1
// KLAMMERN SETZTEN! ->
// #define LEN (30 + 1)
...
char name[LEN*3]; // Achtung: LEN wird vorm Ausrechnen ersetzt.
                  // Also: LEN*3 entspricht 30+1*3 und nicht (30+1)*3
strcpy(name, "Max");
puts(name);
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 ensteht 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)*function(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.:

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

7.6 Bedingte Übersetzung

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

```
#if <const-expr>
#if defined <symbol>
#ifdef <symbol>
#if !defined <symbol>
#ifndef <symbol>
#else
#elif <const-expr>
#endif
```

```
#define MYPUTS puts
...
#ifdef MYPUTS
  MYPUTS("Text");
#else
   printf("Text2");
#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:

```
#ifdef DEBUG
   printf("Debuginformation");
#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.

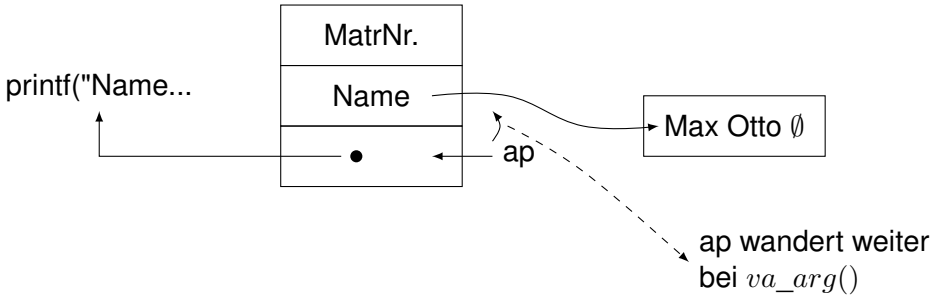
1.) wenigstens ein fester Parameter

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

Macros zum Umgang mit variabler Argumentliste:

```c
#include <stdarg.h> // Voraussetzung für variable Argumentlisten

va_start(ap, la);
// ap: Argument Pointer ( va_list ap; )
// la: Last Argument (letzter Parameter mit Typ und Namen vor ...)
printf("Name: %s, MatrNr: %d \n", Stud.name, Stud.MatrNr);
```




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

```c
va_end(ap);
```

Beispiel: myprintf.c

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

void myprintf(const char* fmt, ...){ // fmt bspw. ("Programm: %s\n", argv[0])
  va_list ap;
  char    *p;
  char    *pstr;
  char    cval;
  int     ival;
  double  dval;

  va_start(ap,fmt);

  for ( p=fmt; // p auf Anfang des Formatsteuerstrings
        *p;
        p++){
    if (*p!='%'){
      myputc(*p);
    } else {
      switch(*++p){
        case 's':
          pstr=va_arg(ap,char*);
          myputs(pstr);
          break;
        case 'd':
          ival=va_arg(ap,int);
          myputd(ival);
          break;
        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 komplierte 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
```