Programmierung I

Vorlesungsskript

Falk Jonatan Strube

Vorlesung von Prof. Dr.-Ing. Beck

25. November 2015

Inhaltsverzeichnis

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.

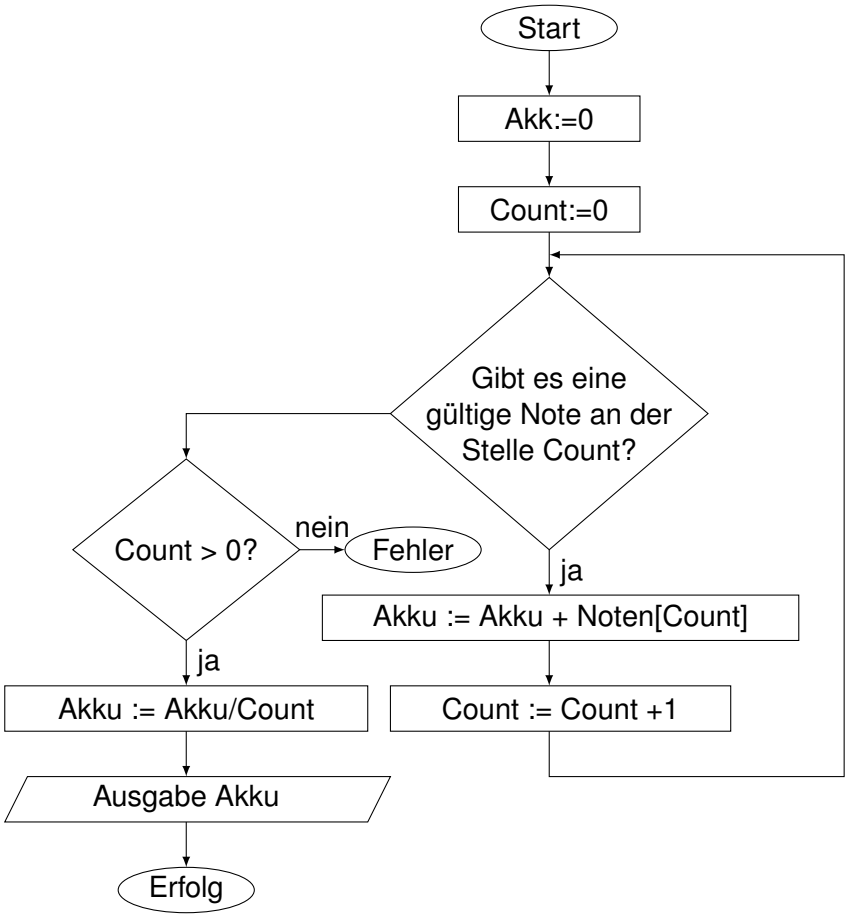
4.) Addiere markierte Zahl zu Akku $\rightarrow$ 5.

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

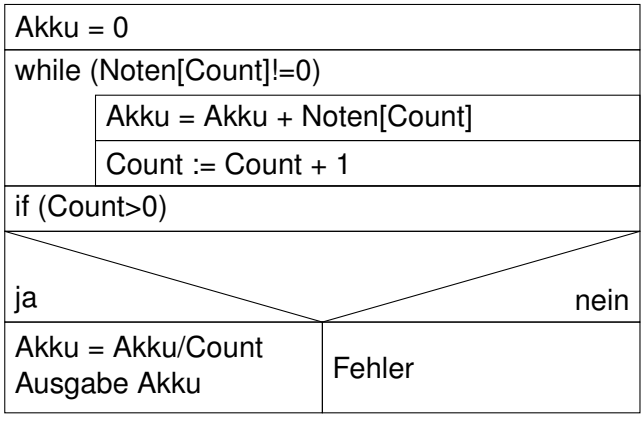
6.) Dividiere Wert in Akku durch Wert in Counter und speichere Akku $\rightarrow$ 7.

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

1.2 Programmablaufplan (PAP)



1.3 Struktogramm / Nassi-Shneiderman-Diagramm



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 Datentypen

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

```
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 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){
    summand = summand *x/i;
    y += summand;
    printf("Summand %d: %lf\n", i, summand);
    i++;
  }
```

```
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 }
```

```
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

```
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];
    printf("Eingabe Zahl: ");
    fgets(buf, 128, stdin);
    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;
}
```

3.7 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)
```

3.8 Header-File

enthält die Funktionsköpfe

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

3.9 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
    Akku = Akku+Noten[Count];
    // Alternativ: (Pointer-Dereferenzierung und Pointer-Arithmetik)
    // * hier keine Operation, sondern der Dereferenzierungs-Stern
    // Akku = Akku + *(Noten+Count);
    // 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  }
```

3.10 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 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.