Teil 3: Aufbau eines C-Programms

Gliederung

Grundelemente eines C-Programms

Zinsberechnung

Eigene Funktionen

Module und Makefiles

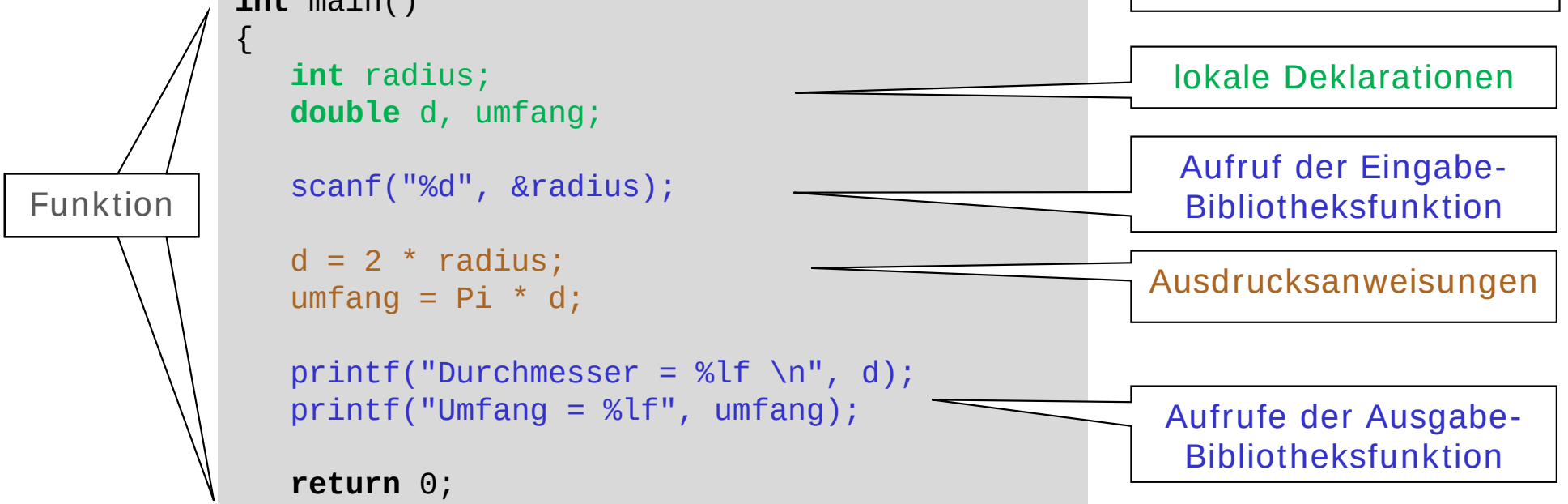
Grundelemente eines C-Programms

Beispielprogramm: Kreisumfang

```
/* Programm zur Berechnung des Kreisumfangs
   eines vom Benutzer einzugebenden Radius */

#include <stdio.h>

const double Pi = 3.14159;

int main()
{
   int radius;
   double d, umfang;

   scanf("%d", &radius);

   d = 2 * radius;
   umfang = Pi * d;

   printf("Durchmesser = %lf \n", d);
   printf("Umfang = %lf", umfang);

   return 0;
}
```

Beispielprogramm: Kreisumfang

```
/* Programm zur Berechnung des Kreisumfangs
   eines vom Benutzer einzugebenden Radius */

#include <stdio.h>

const double Pi = 3.14159;

int main()
{
   int radius;
   double d, umfang;

   scanf("%d", &radius);

   d = 2 * radius;
   umfang = Pi * d;

   printf("Durchmesser = %lf \n", d);
   printf("Umfang = %lf", umfang);

   return 0;
}
```



Deklarationsteil: Variablen

Deklaration:

```
int a, b;          // Ganzzahlvariablen
double c;          // Zahl mit Nachkommastellen
char d;            // Zeichen aus dem ASCII-Code
char wort[6];      // Zeichenkette aus dem ASCII-Code
```

Deklaration mit Initialisierung:

```
int a = 1, b = 99;
double c = 12.3456789;
char d = 'A';
char wort[6] = "Hallo";
```

Fließkomma-Konstanten

- Fließkomma-Konstanten werden standardmäßig als Typ **`double`** gespeichert

```
1.0
24.              // abschliessende Null kann entfallen
.213             // fuehrende Null kann auch entfallen
3.14159
2.123F           // explizit als float speichern
2.2L             // explizit als long double speichern
```

- Exponenten-Schreibweise: vor dem **E** (bzw. **e**) die Mantisse, dahinter der Exponent zur Basis 10

```
2.3e-12
1324.E22
2.45E2F
```

Automatische Initialisierung von Variablen

```
/* Beispiel 1: Ausgabe einer automatisch initialisierten Variablen */

double wert;

int main()
{
    printf("%g\n", wert);
    wert = 1;
    printf("%g\n", wert);
    return 0;
}

/* Beispiel 2: Ausgabe einer nicht initialisierten Variablen */

int main()
{
    double wert;

    printf("%g\n", wert);
    wert = 1;
    printf("%g\n", wert);
    return 0;
}
```

Namenskonventionen für Bezeichner

- Folge von Buchstaben, Ziffern und Unterstrich _
- erstes Zeichen keine Ziffer
- Groß- und Kleinbuchstaben (case-sensitive)
- keine Umlaute
- C99-Standard: Compiler verarbeiten mindestens 31 signifikante Zeichen
- keine Schlüsselwörter der Sprache C:

```
auto        break       case        char        const       continue
default     do          double      else        enum        extern
float       for         goto        if          int         long
register    return      short       signed      sizeof      static
struct      switch      typedef     union       unsigned    void
volatile    while
```

- jedoch erlaubt: **for**m, f**else**n, **do**se

Bibliotheksfunktionen zur Ein-/Ausgabe

Header:

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

Syntax:

```
printf("formatstring", argument1, argument2, ...);
scanf("formatstring", &argument1, &argument2, ...);
```

Beispiele:

```
int zahl;
char buchstabe;
char text[20];

scanf("%d", &zahl);
scanf("%c", &buchstabe);
scanf("%s", text);

printf("Eingegebene Zahl: %d", zahl);
printf("Eingegebener Buchstabe: %c", buchstabe);
printf("Eingegebener Text: %s", text);
```

Eingabe

- Gleiche Formatbezeichner wie bei `printf()` Funktion
- An zweiter Stelle steht die Adresse der Variablen, ermittelbar über den Adress-Operator **&**

```
#include <stdio.h>

int main()
{
  int zahl;

  printf("Bitte Ganzzahl eingeben:\n");
  scanf("%d", &zahl);
  printf("Eingegebene Zahl: %d\n", zahl);

  return 0;
}
```

Kombinationen von Aus-/Eingaben sind **nicht** möglich:

```
scanf("Bitte Zahl eingeben: %d", &zahl);
```

Beispiel: Ein-/Ausgabe

```
/* Einlesen zweier Fliesskommazahlen und deren Addition */

#include <stdio.h>

int main ()
{
    double zahl1, zahl2, ergebnis;

    printf("\nBitte geben sie zwei Fließkommazahlen ein \
    [z.B. 2.34 , 5.23]: ");

    scanf("%lf %lf", &zahl1, &zahl2);

    ergebnis = zahl1 + zahl2;

    printf("Die Summe von %lf und %lf ergibt %lf.\n", \
    zahl1, zahl2, ergebnis);

    return 0;
}
```

Steuerzeichen

\n	Zeilenumbruch
\t	Tabulator horizontal
\b	Backspace: Der Cursor springt ein Zeichen zurück ohne dieses zu löschen
\r	Carriage Return: Cursor springt an den Anfang der aktuellen Zeile
\a	Gibt einen Peep-Ton aus
\\	Gibt den Backslash selbst aus
\"	Gibt ein " aus
\'	Gibt ein ' aus
\?	Gibt ein ? aus

Häufig gebrauchte Formatbezeichner

%i	Dezimalzahl ausgeben
%d	Dezimalzahl ausgeben
%x	Hexadezimalzahl mit kleinen Buchstaben
%X	Hexadezimalzahl mit großen Buchstaben
%e	Zahl in Exponentialdarstellung ausgeben
%u	Vorzeichenlose Zahl ausgeben
%c	Zeichen ausgeben
%s	Zeichenkette ausgeben
%f	Fließkommazahl (float) ausgeben
%lf	Fließkommazahl (double) ausgeben

Sonderzeichen

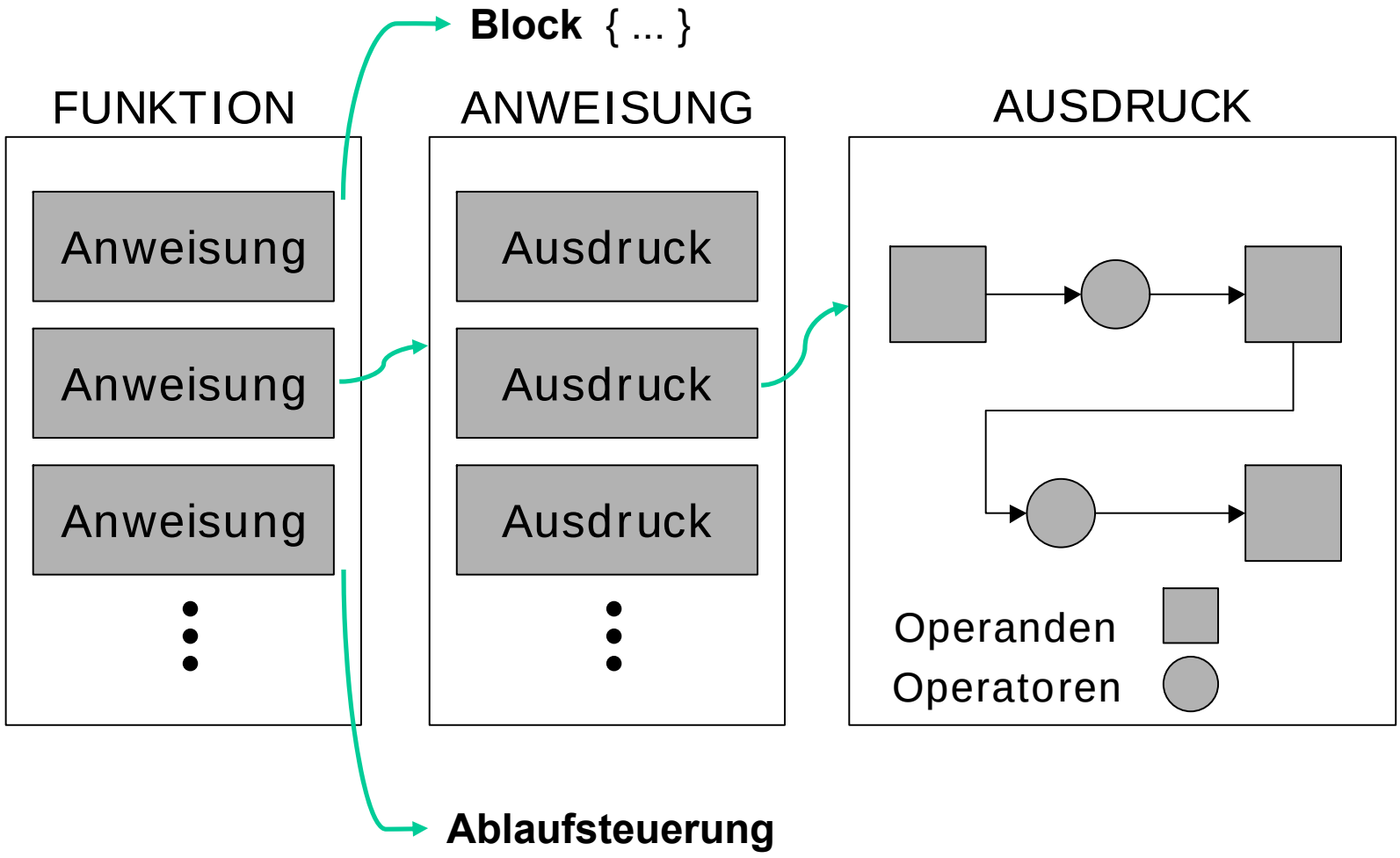
- Umlaute und Sonderzeichen müssen über ihren ASCII Code ausgegeben werden

```
#include <stdio.h>

int main()
{
    printf("%c\n", 148);
    printf("%d\n", 148);

    return 0;
}
```

```
ö
148
```

Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char
0	00		43	2B	+	86	56	V	129	81	ü	172	AC	¼	215	D7	╫
1	01	☺	44	2C	,	87	57	W	130	82	é	173	AD	¡	216	D8	╪
2	02	☻	45	2D	-	88	58	X	131	83	â	174	AE	«	217	D9	┘
3	03	♥	46	2E	.	89	59	Y	132	84	ä	175	AF	»	218	DA	┌
4	04	♦	47	2F	/	90	5A	Z	133	85	à	176	B0	░	219	DB	█
5	05	♣	48	30	0	91	5B	[	134	86	å	177	B1	▒	220	DC	▄
6	06	♠	49	31	1	92	5C	\	135	87	ç	178	B2	▓	221	DD	▌
7	07	•	50	32	2	93	5D	]	136	88	ê	179	B3	│	222	DE	▐
8	08	◘	51	33	3	94	5E	^	137	89	ë	180	B4	┤	223	DF	▀
9	09	○	52	34	4	95	5F	_	138	8A	è	181	B5	╡	224	E0	α
10	0A	◙	53	35	5	96	60	'	139	8B	ï	182	B6	╢	225	E1	β
11	0B	♂	54	36	6	97	61	a	140	8C	î	183	B7	╖	226	E2	Γ
12	0C	♀	55	37	7	98	62	b	141	8D	ì	184	B8	╕	227	E3	π
13	0D	♪	56	38	8	99	63	c	142	8E	Ä	185	B9	╣	228	E4	Σ
14	0E	♫	57	39	9	100	64	d	143	8F	Å	186	BA	║	229	E5	σ
15	0F	☼	58	3A	:	101	65	e	144	90	É	187	BB	╗	230	E6	µ
16	10	►	59	3B	;	102	66	f	145	91	æ	188	BC	╝	231	E7	τ
17	11	◄	60	3C	<	103	67	g	146	92	Æ	189	BD	╜	232	E8	Φ
18	12	↕	61	3D	=	104	68	h	147	93	ô	190	BE	╛	233	E9	Θ
19	13	‼	62	3E	>	105	69	i	148	94	ö	191	BF	┐	234	EA	Ω
20	14	¶	63	3F	?	106	6A	j	149	95	ò	192	C0	└	235	EB	δ
21	15	§	64	40	@	107	6B	k	150	96	û	193	C1	┴	236	EC	∞
22	16	▬	65	41	A	108	6C	l	151	97	ù	194	C2	┬	237	ED	Ø
23	17	↨	66	42	B	109	6D	m	152	98	ÿ	195	C3	├	238	EE	ε
24	18	↑	67	43	C	110	6E	n	153	99	Ö	196	C4	─	239	EF	∩
25	19	↓	68	44	D	111	6F	o	154	9A	Ü	197	C5	┼	240	F0	≡
26	1A	→	69	45	E	112	70	p	155	9B	¢	198	C6	╞	241	F1	±
27	1B	←	70	46	F	113	71	q	156	9C	£	199	C7	╟	242	F2	≥
28	1C	└	71	47	G	114	72	r	157	9D	¥	200	C8	╚	243	F3	≤
29	1D	↔	72	48	H	115	73	s	158	9E	Pts	201	C9	╔	244	F4	⌠
30	1E	▲	73	49	I	116	74	t	159	9F	ƒ	202	CA	╩	245	F5	⌡
31	1F	▼	74	4A	J	117	75	u	160	A0	á	203	CB	╦	246	F6	÷
32	20		75	4B	K	118	76	v	161	A1	í	204	CC	╠	247	F7	≈
33	21	!	76	4C	L	119	77	w	162	A2	ó	205	CD	═	248	F8	°
34	22	"	77	4D	M	120	78	x	163	A3	ú	206	CE	╬	249	F9	•
35	23	#	78	4E	N	121	79	y	164	A4	ñ	207	CF	╧	250	FA	·
36	24	$	79	4F	O	122	7A	z	165	A5	Ñ	208	D0	╨	251	FB	√
37	25	%	80	50	P	123	7B	{	166	A6	ª	209	D1	╤	252	FC	ⁿ
38	26	&	81	51	Q	124	7C	|	167	A7	º	210	D2	╥	253	FD	²
39	27	'	82	52	R	125	7D	}	168	A8	¿	211	D3	╙	254	FE	■
40	28	(	83	53	S	126	7E	~	169	A9	⌐	212	D4	╘	255	FF	
41	29	)	84	54	T	127	7F	⌂	170	AA	¬	213	D5	╒			
42	2A	*	85	55	U	128	80	Ç	171	AB	½	214	D6	╓			

Funktion – Anweisung – Ausdruck



Arten von Anweisungen

Block { ... }

- durch geschweifte Klammern eingeschlossen
- jede Anweisung, die kein Block ist, wird durch Semikolon terminiert

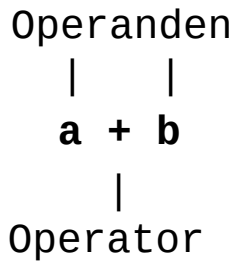
Ausdrucksanweisung

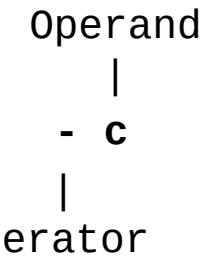
Zuweisung:	`m = n + 2;`
Funktionsaufruf	`printf("Hallo");`
Leere Anweisung	`;`

Ablaufsteuerung

Auswahlanweisung
Wiederholungsanweisung
Sprunganweisung

Ausdrücke





Ein **Ausdruck** ist eine Folge von Operanden, Operatoren und möglichen Klammern.

Eine **Ausdrucksanweisung** hat **immer** einen Rückgabewert eines bestimmten Typs (im Gegensatz zu Blöcken oder Ablaufsteuerungen).

Operatoren

- Anwendung auf einen oder mehrere Operanden (unär, binär)
- Bildung von (Ergebnis-)Werten
- mögliche Nebeneffekte sind zu berücksichtigen

```
()   []   ->   .
!    ~    ++   --   +    -    *    &    (Typname)   sizeof
/    %    <<   >>   <    <=   >    >=   ==   !=   ^    |    &&
||   ?:   =    +=   -=   *=   /=   %=   &=   ^=   |=   <<=  >>=  ,
```

Vorrangregeln bei Operatoren

Operatorklasse	Operatoren	Assoziativität
unär	! ~ ++ -- + –	von rechts nach links
multiplikativ	* / %	von links nach rechts
additiv	+ -	von links nach rechts
shift	<< >>	von links nach rechts
relational	< <= > >=	von links nach rechts
Gleichheit	== !=	von links nach rechts
bitweise	&	von links nach rechts
bitweise	^	von links nach rechts
bitweise	|	von links nach rechts
logisch	&&	von links nach rechts
logisch	||	von links nach rechts
Bedingte Bewertung	?:	von rechts nach links
Zuweisung	= op=	von rechts nach links
Reihung	,	von links nach rechts

Arithmetikoperatoren

Klasse	Operatoren	Beispiele
Unär	**-**	`-a, -(x + 1)`

Binär	**+ , - , * , / , %**	`a + b`
		`a % d`
		`a * b / c`

Relationale Operatoren

Klasse	Operatoren	Beispiele
relational	**< , > , <= , >=**	`a < b` `x >= d`
Gleichheit	**==** **!=**	`a == b` `c != d`

Ausdrücke mit relationalen Operatoren liefern als Ergebnis den Typ **int** zurück, mit den möglichen Werten **0** (false) und **!= 0** (true).

Logische Operatoren

AND (&&)			OR (||)			NOT (!)	
e1 && e2			**e1 || e2**			**! e1**	
e1	e2	=	e1	e2	=	e1	=
T	*T*		*T*	*T*		*T*	
T	*F*		*T*	*F*		*F*	
F	*T*		*F*	*T*			
F	*F*		*F*	*F*			

Beispiele:

```
if (zahl < 10 || zahl > 20))  { ... }
if (!(x1 && x2 && x3)  { ... }
```

ist logisch äquivalent zu

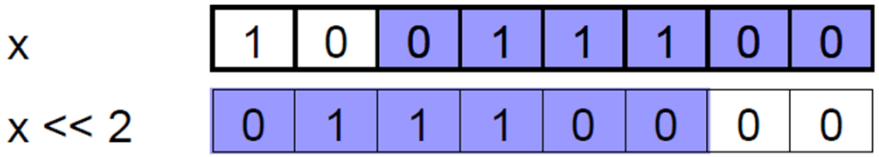
```
if (!x1 || !x2 || !x3)  { ... }
```

Bitweise Operatoren

- Typische Anwendung: Status, Fehlerzustände, Signale

Symbol	Bedeutung
~	Negation
<< , >>	left shift (Bitweises Linksschieben), right shift (Bitweises Rechtsschieben)
& , | , ^	AND , OR , XOR

- Beispiel: Operation "left shift"



Bitweise Operatoren

- Typische Anwendung: Status, Fehlerzustände, Signale

Symbol	Bedeutung
~	Negation
<< , >>	left shift (Bitweises Linksschieben), right shift (Bitweises Rechtsschieben)
& , | , ^	AND , OR , XOR

x	0 0 1 1
y	1 0 1 0
~x	1 1 0 0
x & y	0 0 1 0
x | y	1 0 1 1
x ^ y	1 0 0 1
x << 2	1 1 0 0
y >> 2	0 0 1 0

Inkement und Dekrement

- Der Inkement-Operator ++ bedeutet "das nächste":

```
int x;
x++;   // entspricht   x = x + 1;
```

- Dekrement Operator -- bedeutet "das vorherige":

```
x--;   // entspricht   x = x - 1;
```

- Beispiel:

```
int x;
x = 3;
z = x++ - 2;
```

POSTFIX

z = 1

```
int x;
x = 3;
z = ++x - 2;
```

PREFIX

z = 2

Zinsberechnung

Beispiel: Zinsberechnung

- Berechnung der jährlichen Entwicklung eines Grundkapitals über eine vorgegebene Laufzeit
- Zinsen werden mit dem Kapital wieder angelegt
- Erzeugung einer Tabelle mit folgenden Angaben:
  - laufendes Jahr und angesammeltes Kapital (in EUR)
  - Laufzeit: 10 Jahre
  - Grundkapital: 1000 EUR
  - Zins: 5%

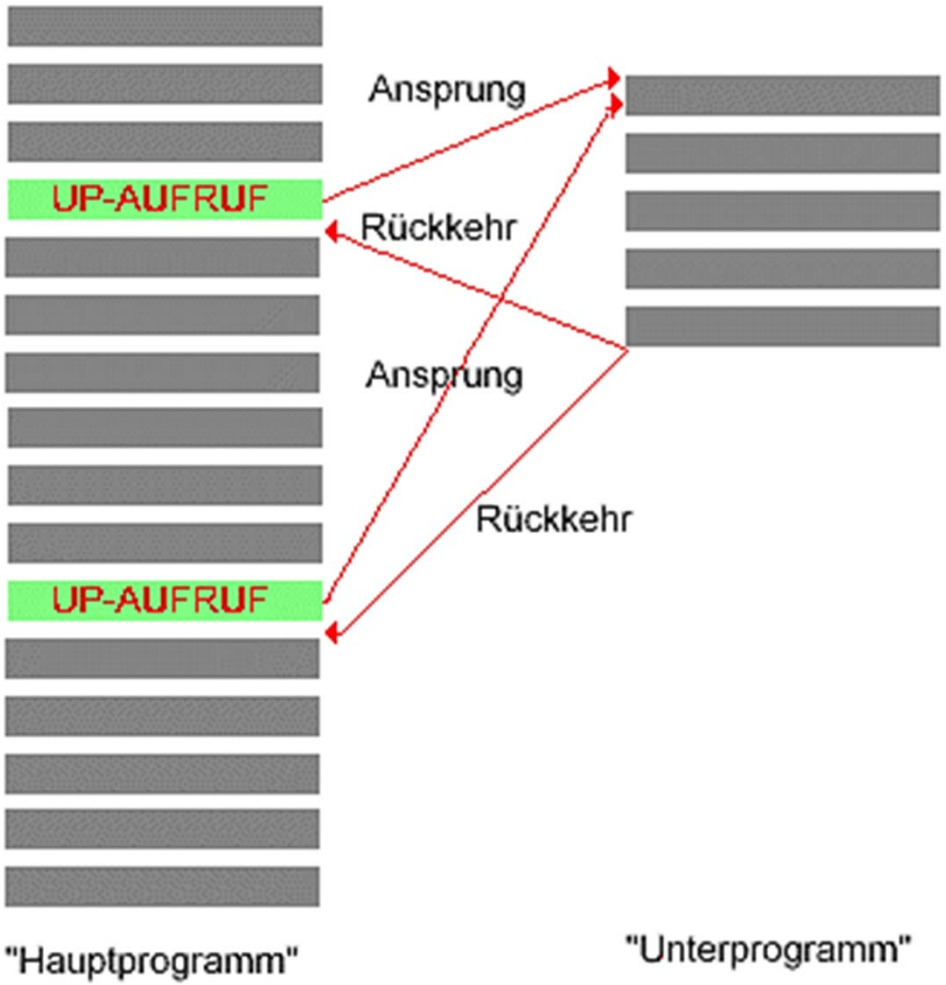
```c
/* Anwendung zur Zinsberechnung */
#include <stdio.h>

#define LAUFZEIT 10
#define GRUNDKAPITAL 1000.
#define ZINS 5

int main()
{
    int jahr;
    double kapital = GRUNDKAPITAL;

    printf ("Zinstabelle fuer Grundkapital %7.2f EUR\n", kapital );
    printf ("Kapitalstand zum Jahresende:\n");
    for (jahr = 1 ; jahr <= LAUFZEIT ; jahr = jahr++ )
    {
        printf ("\nJahr: %2d\t", jahr);
        kapital = kapital * (1. + ZINS / 100 );
        printf ("Kapital: %7.2f EUR", kapital );
    }
    printf ("\n\nAus %7.2f EUR Grundkapital\n", GRUNDKAPITAL );
    printf ("wurden in %d Jahren %7.2f EUR\n", LAUFZEIT , kapital );
    return 0;
}
```

```c
/* Anwendung zur Zinsberechnung */
#include <stdio.h>

#define LAUFZEIT 10
#define GRUNDKAPITAL 1000.00
#define ZINS 5.0

int main()
{
    int jahr;
    double kapital = GRUNDKAPITAL;

    printf ("Zinstabelle fuer Grundkapital %7.2f EUR\n", GRUNDKAPITAL);
    printf ("Kapitalstand zum Jahresende:\n");
    for (jahr = 1; jahr <= LAUFZEIT; jahr = jahr + 1)
    {
        printf ("\nJahr: %2d\t", jahr);
        kapital = kapital * (1. + ZINS / 100.);
        printf ("Kapital: %7.2f EUR", kapital);
    }
    printf ("\n\nAus %7.2f EUR Grundkapital\n", GRUNDKAPITAL);
    printf ("wurden in %d Jahren %7.2f EUR\n", LAUFZEIT, kapital);
    return 0;
}
```

```
Zinstabelle fuer Grundkapital 1000.00 EUR
Kapitalstand zum Jahresende:

Jahr:  1         Kapital: 1050.00 EUR
Jahr:  2         Kapital: 1102.50 EUR
Jahr:  3         Kapital: 1157.62 EUR
Jahr:  4         Kapital: 1215.51 EUR
Jahr:  5         Kapital: 1276.28 EUR
Jahr:  6         Kapital: 1340.10 EUR
Jahr:  7         Kapital: 1407.10 EUR
Jahr:  8         Kapital: 1477.46 EUR
Jahr:  9         Kapital: 1551.33 EUR
Jahr: 10         Kapital: 1628.89 EUR

Aus 1000.00 EUR Grundkapital
wurden in 10 Jahren 1628.89 EUR
```

Eigene Funktionen

Hauptprogramm - Unterprogramm



Funktionen

- Umsetzung der Unterprogrammtechnik
- Lösung von Teilproblemen, Gruppierung häufig benötigter Anweisungsfolgen
- Datenabhängigkeit durch Parameterübergabe
- Aufbau:

Funktionskopf

```
Rückgabetyp Funktionsname(typ1 bezeichner1, typ2 bezeichner2, ...)
{
    lokale Deklarationen;
    ...
    Anweisungen;
    ...
    return Ausdruck;  // Ausdruck muss vom Typ Rückgabetyp sein
}
```

Funktionsrumpf

Übergabeparameter und Rückgabewert

```
Rückgabetyp Funktionsname(typ1 parameter1, typ2 parameter2, ...)
{
        Lokale Deklarationen;

        Anweisung1;
        Anweisung2;
        ...
        return Ausdruck;
}
```

Wertübergabe **Hauptprogramm -> Unterprogramm**

- bei Funktionsaufruf
- Funktionsparameter in beliebiger Anzahl

Wertübergabe **Unterprogramm -> Hauptprogramm**

- bei Beendigung des Unterprogramms
- maximal 1 Wert

Aufruf einer Funktionen

```
int main()
{
  int max, x = 2, y = 3;
  max = maximum(x, y);
  printf("Das Maximum aus %d und %d ist %d.", x, y, max);
  return 0;
}
```

Aufruf einer Funktionen

```
int main()
{
  int max, x = 2, y = 3;
  max = maximum(x, y);
  printf("Das Maximum aus %d und %d ist %d.", x, y, max);
  return 0;
}

int maximum(int a, int b)
{

}
```

Aufruf einer Funktionen

```
int main()
{
  int max, x = 2, y = 3;
  max = maximum(x, y);
  printf("Das Maximum aus %d und %d ist %d.", x, y, max);
  return 0;
}

int maximum(int a, int b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

Aufruf einer Funktionen

```
int maximum(int, int);     // Funktions-Prototyp

int main()
{
  int max, x = 2, y = 3;
  max = maximum(x, y);
  printf("Das Maximum aus %d und %d ist %d.", x, y, max);
  return 0;
}

int maximum(int a, int b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

Aufruf einer Funktionen

```
int maximum(int a, int b);     // Funktions-Prototyp,
                               // Bezeichner sind hier optional
int main()
{
  int max, x = 2, y = 3;
  max = maximum(x, y);
  printf("Das Maximum aus %d und %d ist %d.", x, y, max);
  return 0;
}

int maximum(int a, int b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

Deklaration vs. Definition

Deklaration:

Entspricht dem Prototyp und enthält den Funktionskopf

```
typ Funktionsname (typ1, typ2, ...);
typ Funktionsname (typ1 parameter1, typ2 parameter2, ...);
```

Definition:

Entspricht der Implementierung und enthält Funktionskopf + Funktionsrumpf

```
typ Funktionsname (typ1 parameter1, typ2 parameter2, ...)
{
      ...
      return Ausdruck;
}
```

Beispiel

```c
int main()
{
    int untere, obere, resultat;

    printf("Eingabe der unteren und oberen Grenze: ");
    scanf("%d %d", &untere, &obere);
    resultat = summe(untere, obere);
    printf("Die Summe der Zahlen von %d bis %d ist %d\n",
            untere, obere, resultat);
    return 0;
}

int summe(int u, int o)
{
    int sum = 0;
    for (int i = u; i <= o; i++)
    {
     sum += i;
    }
    return sum;
}
```

Module

Programm aus mehreren Quelldateien

modul1.c

```
#include <stdio.h>

int main()
{
  int u;

  i = 1;
  abc();
  u = sum(17, 4);
  printf("%d", u);
  return 0;
}
```

modul2.c

```
int i;

int sum(int x, int y)
{
  return x + y;
}
```

modul3.c

```
void abc()
{
  i = 10;
}
```

Programm aus mehreren Quelldateien

modul1.c

```
#include <stdio.h>


extern int i;

int main()
{
  int u;

  i = 1;
  abc();
  u = sum(17, 4);
  printf("%d", u);
  return 0;
}
```

modul2.c

```
int i;

int sum(int x, int y)
{
  return x + y;
}
```

modul3.c

```
extern int i;

void abc()
{
  i = 10;
}
```

Programm aus mehreren Quelldateien

modul1.c

```
#include <stdio.h>

int sum(int, int);
void abc();
extern int i;

int main()
{
  int u;

  i = 1;
  abc();
  u = sum(17, 4);
  printf("%d", u);
  return 0;
}
```

modul2.c

```
int i;

int sum(int x, int y)
{
  return x + y;
}
```

modul3.c

```
extern int i;

void abc()
{
  i = 10;
}
```

Programm aus mehreren Quelldateien

modul2.h

```
extern int i;

int sum(int, int);
```

modul3.h

```
void abc();
```

modul1.c

```
#include <stdio.h>

int main()
{
  int u;

  i = 1;
  abc();
  u = sum(17, 4);
  printf("%d", u);
  return 0;
}
```

modul2.c

```
int i;

int sum(int x, int y)
{
  return x + y;
}
```

modul3.c

```
void abc()
{
  i = 10;
}
```

Programm aus mehreren Quelldateien

modul2.h

```
extern int i;

int sum(int, int);
```

modul3.h

```
void abc();
```

modul1.c

```
#include <stdio.h>
#include "modul2.h"
#include "modul3.h"

int main()
{
  int u;

  i = 1;
  abc();
  u = sum(17, 4);
  printf("%d", u);
  return 0;
}
```

modul2.c

```
int i;

int sum(int x, int y)
{
  return x + y;
}
```

modul3.c

```
#include "modul2.h"

void abc()
{
  i = 10;
}
```

Programm aus mehreren Quelldateien

modul2.h

```
extern int i;

int sum(int, int);
```

modul3.h

```
void abc();
```

modul1.c

```
#include <stdio.h>
#include "modul2.h"
#include "modul3.h"

int main()
{
  int u;

  i = 1;
  abc();
  u = sum(17, 4);
  printf("%d", u);
  return 0;
}
```

modul2.c

```
#include "modul2.h"

int i;

int sum(int x, int y)
{
  return x + y;
}
```

modul3.c

```
#include "modul2.h"
#include "modul3.h"

void abc()
{
  i = 10;
}
```

Übersetzungssteuerung mit Makefiles

Definitionen von Variablen: `CC = gcc –Wall`

Kommentare: `# Dies ist ein Kommentar!`

Includes: `-include Makefile.local`

Regeln: `%.o: %.c; -CC $<`

Syntax für Regeln:

Target [weitere **Targets**]:[:] [**Vorbedingungen**] [; **Kommandos**]
[<Tab> **Kommandos**]
[<Tab> **Kommandos**]

Beispiel

modul2.h

```
extern int i;

int sum(int, int);
```

modul3.h

```
void abc();
```

modul1.c

```
#include <stdio.h>
#include "modul2.h"
#include "modul3.h"

int main()
{
  int u;

  i = 1;
  abc();
  u = sum(17, 4);
  printf("%d", u);
  return 0;
}
```

modul2.c

```
#include "modul2.h"

int i;

int sum(int x, int y)
{
  return x + y;
}
```

modul3.c

```
#include "modul2.h"
#include "modul3.h"

void abc()
{
  i = 10;
}
```

```
01 CC = gcc –Wall
02 Objektdateien = modul1.o modul2.o modul3.o
03 Programm = hallo.exe
04
05 $(Programm): $(Objektdateien)
06      $(CC) -o $@ $^
07
08 # explizite Regel definiert das Kommando
09 $(Objektdateien):
10      $(CC) -c $<
11
12 # implizite Regeln definieren die Abhängigkeiten
13 modul1.o: modul1.c modul2.h modul3.h
14 modul2.o: modul2.c
15 modul3.o: modul3.c
16
17 all: clean $(Programm) run
18
19 clean:
20      rm $(Objektdateien) $(Programm) -f
21
22 run:
23      ./$(Programm)
```

Target *[weitere Targets]:[:] [**Vorbedingungen**]*

[<Tab> ***Kommandos****]*

```
modul1.o: modul1.c
	gcc -c modul1.c
```

Target *[weitere Targets]:[:] [**Vorbedingungen**]*

[<Tab> ***Kommandos****]*

```
modul1.o: modul1.c
	gcc -c modul1.c

hallo.exe: modul1.o
	gcc -o hallo.exe modul1.o
```

Target *[weitere Targets]:[:]* ***[Vorbedingungen]***

[<Tab> ***Kommandos****]*

```
hallo.exe: modul1.o
        gcc -o hallo.exe modul1.o

modul1.o: modul1.c
        gcc -c modul1.c
```

Die Regel für die Programmdatei steht an erster Stelle, damit die Abhängigkeiten auch dann funktionieren, wenn beim Aufruf von "make" kein Target angegeben wird.

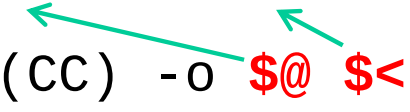
```
CC = gcc -Wall
hallo.exe: modul1.o
	$(CC) –o hallo.exe modul1.o
modul1.o: modul1.c
	$(CC) –c modul1.c
```

Variable: CC

```
CC = gcc -Wall
hallo.exe: modul1.o
        $(CC) -o $@ modul1.o
modul1.o: modul1.c
        $(CC) –c modul1.c
```

Variable: CC
Target-Name: **$@**

```
CC = gcc -Wall

hallo.exe: modul1.o
        $(CC) -o $@ $<

modul1.o: modul1.c
        $(CC) -c $<
```



Variable:	CC
Target-Name:	**$@**
erste Vorbedingung:	**$<**

```
01 CC = gcc -Wall
02
03 hallo.exe: modul1.o modul2.o modul3.o
04     $(CC) -o $@ $^
05
06 modul1.o: modul1.c modul2.h modul3.h
07     $(CC) -c $<
08
09 modul2.o: modul2.c modul2.h
10     $(CC) -c $<
11
12 modul3.o: modul3.c modul3.h modul2.h
13     $(CC) -c $<
```

Targetname:	**$@**
erste Vorbedingung:	**$<**
alle Vorbedingungen:	**$^**

```
01 CC = gcc -Wall
02
03 hallo.exe: modul1.o modul2.o modul3.o
04     $(CC) -o $@ $^
05
06 modul1.o: modul1.c modul2.h modul3.h
07     $(CC) -c $<
08
09 modul2.o: modul2.c modul2.h
10     $(CC) -c $<
11
12 modul3.o: modul3.c modul3.h modul2.h
13     $(CC) -c $<
14
15 all: clean hallo.exe run
16
17 clean:
18     rm *.o hallo.exe -f
19
20 run:
21     ./hallo.exe
```

```
01 CC = gcc -Wall
02
03 hallo.exe: modul1.o modul2.o modul3.o
04     $(CC) -o $@ $^
05
06 modul1.o: modul1.c modul2.h modul3.h
07     $(CC) -c $<
08
09 modul2.o: modul2.c modul2.h
10     $(CC) -c $<
11
12 modul3.o: modul3.c modul3.h modul2.h
13     $(CC) -c $<
14
15 all: clean hallo.exe run
16
17 clean:
18     rm *.o hallo.exe -f
19
20 run:
21     ./hallo.exe
```

```
01 CC = gcc -Wall
02
03 hallo.exe: modul1.o modul2.o modul3.o
04     $(CC) -o $@ $^
05
06 # explizite Regel definiert das Kommando
07 modul1.o modul2.o modul3.o:
08     $(CC) -c $<
09
10 # implizite Regeln definieren die Abhaengigkeiten
11 modul1.o: modul1.c modul2.h modul3.h
12 modul2.o: modul2.c modul2.h
13 modul3.o: modul3.c modul3.h modul2.h
14
15 all: clean hallo.exe run
16
17 clean:
18     rm *.o hallo.exe -f
19
20 run:
21     ./hallo.exe
```

```
01 CC = gcc -Wall
02
03 hallo.exe: modul1.o modul2.o modul3.o
04     $(CC) -o $@ $^
05
06 # explizite Regel definiert das Kommando
07 modul1.o modul2.o modul3.o:
08     $(CC) -c $<
09
10 # implizite Regeln definieren die Abhaengigkeiten
11 modul1.o: modul1.c modul2.h modul3.h
12 modul2.o: modul2.c modul2.h
13 modul3.o: modul3.c modul3.h modul2.h
14
15 all: clean hallo.exe run
16
17 clean:
18     rm *.o hallo.exe -f
19
20 run:
21     ./hallo.exe
```

```
01 CC = gcc –Wall
02 Objektdateien = modul1.o modul2.o modul3.o
03 Programm = hallo.exe
04
05 $(Programm): $(Objektdateien)
06      $(CC) -o $@ $^
07
08 # explizite Regel definiert das Kommando
09 $(Objektdateien):
10      $(CC) -c $<
11
12 # implizite Regeln definieren die Abhängigkeiten
13 modul1.o: modul1.c modul2.h modul3.h
14 modul2.o: modul2.c modul2.h
15 modul3.o: modul3.c modul3.h modul2.h
16
17 all: clean $(Programm) run
18
19 clean:
20      rm $(Objektdateien) $(Programm) -f
21
22 run:
23      ./$(Programm)
```

```
01 CC = gcc –Wall
02 Objektdateien = modul1.o modul2.o modul3.o
03 Programm = hallo.exe
04
05 $(Programm): $(Objektdateien)
06      $(CC) -o $@ $^
07
08 # explizite Regel definiert das Kommando
09 $(Objektdateien):
10      $(CC) -c $<
11
12 # implizite Regeln definieren die Abhängigkeiten
13 modul1.o: modul1.c modul2.h modul3.h
14 modul2.o: modul2.c modul2.h
15 modul3.o: modul3.c modul3.h modul2.h
16
17 all: clean $(Programm) run
18
19 clean:
20      rm $(Objektdateien) $(Programm) -f
21
22 run:
23      ./$(Programm)
```