3. Basisdatentypen
-Teil B-

Motivation

Die Programmiersprache C stellt eine Reihe „eingebauter“ Datentypen zur Verfügung

- ✓ Ganze Zahlen
- ✓ Gleitkommazahlen
- – Zeichen
- – Aufzählungstypen
- – Zeiger

Zeichen

Man kann den Datentyp `char` als Ganzzahl-Datentyp sehen und mit Variablen dieses Typs arbeiten wie mit anderen Ganzzahl-Variablen auch.

```
int main(void)
{
        char x=0;
        x++;
        printf("Ergebnis als  Zahl   %d\n",x);
        printf("Ergebnis als Zeichen %c\n",x);
}
```

```
Ergebnis als    Zahl   1
Ergebnis als Zeichen   ☺
```

Zeichen

Der **eigentliche Zweck** des Datentyps `char` ist aber das **Speichern von Zeichen**.

Zeichen werden nach dem **ASCII-Code** als Zahlenwert von 0 bis 255 verschlüsselt.

ASCII = American Standard Code for Information Interchange

Zeichen

Der Zahlenwert wird in **einem Byte Speicherplatz** abgespeichert.
Die Datentypen einer Zeichen-Variable sind

```
(signed) char
unsigned char
```

Die Umwandlung von Zeichen in Zahlenwerte und umgekehrt erfolgt bei der Ein- und Ausgabe mit der **Formatanweisung "%c"** automatisch.

Zeichen

Zeichen werden nach dem **ASCII-Code** als Zahlenwert von 0 bis 255 verschlüsselt.

ASCII = American Standard Code for Information Interchange

Orginal-Code: 7-Bit Code d.h. 0 ... 127 Zeichen
95 druckbare ASCII Zeichen (32_{10} … 126_{10})

Extended ASCII: 8-Bit d.h. 0 … 255 Zeichen
umfaßt 7-Bit Code + weitere 128 Zeichen

Achtung: nicht standardisiert!

ASCII- Tabelle

0-127

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	SP
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂

ASCII- Tabelle

0-127

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Zeichen

Die Zuweisung eines festen Wertes an eine Variable vom Typ `char` kann auf vielfältige Weise geschehen,
z.B. der Großbuchstabe A

```
char x;
x = 'A';
x = '\101';
x = 65;
x = (char)65;
x = 0101;
x = 0x41;
```

```
printf("%c",x);
```

ergibt in jedem Fall die Ausgabe A

Erinnerung: `\o` numerische Angabe eines Zeichens in **oktalem ASCII-Code,**
wobei an Stelle von `o` eine ein- bis dreistellige Oktalzahl stehen muss
z.B. \0

ASCII- Tabelle

128-255

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	ÿ	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	█	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
141	ì	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	░	192	└	208	╨	224	α	240	≡		

Zeichen

Sofern der Extended Code unterstützt wird, kann man auch die deutschen Umlaute z.B. Ä ausgeben:

```
char x;
x = '\216';
x = 142;
x = (char)142;
x = 0216;
x = 0x8e;
```

```
printf("%c",x);
```

ergibt dann in jedem Fall die Ausgabe Ä

Testen mit Programm: *3-4-char-test-1.c*

Zeichen

Üblicherweise wird der Character-Datentyp mit Vorzeichen verwendet, d.h.

Wertebereich auf dem Zahlenkreis : –128 bis 127.

ASCII-Codewerte über 127 = negativen Zahlen

⇒bei Typumwandlung von `(signed) char` nach `int` muss man dies berücksichtigen!

d.h. einen negativen `int`-Wert um 256 erhöhen, um tatsächlich den ASCII-Code des Zeichens zu erhalten.

3-4-char-test-2.c

```
int main(void)
{
      char x;
      int i, j;

      x = 'Ä';          /* alternativ: '\216' */
      i = (int)x;       /* (int) optional! */
      j = i;

      if (j<0) { j=j+256; }

      printf("Zeichen:          %c\n",x);
      printf("gecastet zu int: %d\n",i);
      printf("ASCII-Code:       %d\n\n",j);
}
```

```
Zeichen:           Ä
gecastet zu int:   -114
ASCII-Code:        142
```

Einlesen eines Zeichens mit `getchar()`

Das **Einlesen eines Zeichens** kann außer mit der Funktion

```
int scanf ( "%c„, &char_var)
```

auch mit der Funktion

```
int getchar()
```

erfolgen.

Die Funktion hat **keinen** Parameter und liefert den ASCII-Code des eingelesenen Zeichens als **`int`**-Wert zurück.

Einlesen eines Zeichens mit `getchar()`

Problem: die im Eingabepuffer übrig gebliebenen Zeichen löschen - insbesondere das nach einer Eingabe dort verbleibende Zeichen mit ASCII-Code 10 (= Return-Taste)

Jede Eingabe schließt mit der Return-Taste d.h. das Zeichen steht also immer am Ende

→ man muss einfach mit Hilfe einer Schleife solange Zeichen aus dem Eingabepuffer lesen, bis man den ASCII-Code 10 (\12) gelesen hat:

```
while (getchar()!='\12');

for (;getchar()!=(char)10;);
```

Einlesen eines Zeichens mit `getchar()`

Per Hand nach jedem `scanf` jeweils ein Schleife, die den Eingabepuffer leert, einzufügen, ist sehr mühsam:
Daher definiert man **ein Makro**

```
#define INCLR while (getchar()!='\12');
```

Dieses Makro (=Abkürzung) kann einfach nach jedem Aufruf einer Eingabefunktion eingefügt werden:

```
scanf(…); INCLR
```

Der **Präprozessor** ersetzt automatisch das `INCLR` durch die oben definierte Schleife und zwar ***bevor der Compiler*** das Programm übersetzt!

Einlesen eines Zeichens mit `getchar()`

Das Makro INCLR steht in der Datei *Diverses.h*, die mit

```
#include "Diverses.h"
```

in jedes Programm eingefügt werden kann!

Der Einsatz der Funktion `getchar()` und des Makros `INCLR` wird im folgenden Beispielprogramm gezeigt, das nicht wegen Fehleingaben in eine Endlosschleife oder zu einem Abbruch kommen kann.

3-4-char-test-3.c

```
#include "Diverses.h"

int main(void){
    char weiter;
    int r, i;
    /* Wiederholungsschleife */
    do  {
        /* Schleife fuer Eingabeueberpruefung */
        do {
            printf("\nGib ASCII-Code ein (0-255): ");
            r=scanf("%d",&i); INCLR
        }
        while (r<1 || i<0 || i>255);

        printf("Das Zeichen ist %c\n\n",(char)i);
        printf("Nochmal? (j/n) ");
        weiter=getchar(); INCLR

    }  while (weiter=='j' || weiter=='J');
}
```

Aufzählungstypen - `enum`

Man kann neue Variablentypen als Aufzählungstypen definieren, deren Wertemenge man selbst festlegen kann:

```
enum farbe {rot, blau, grau, schwarz};
enum farbe f, g, h;
```

oder alternativ

```
typedef enum {rot, blau, grau, schwarz} farbe;
farbe f, g, h;
```

Die Werte werden intern in der Listenreihenfolge als Ordinalzahlen repräsentiert:

0, 1, 2, 3, …

Aufzählungstypen - `enum`

Zur Speicherung einer Variablen vom Aufzählungstypen werden 4 Bytes benötigt:

```
printf("%i",sizeof(farbe));                4
```

```
printf("%i",sizeof(enum farbe));           4
```

Der sizeof-Operator funktioniert natürlich auch für andere Datentypen:

```
printf("%i",sizeof(int));                  4
```

```
printf("%i",sizeof(float));                4
```

Aufzählungstypen - `enum`

Das direkte **Einlesen** eines Wertes einer Variablen eines Auszählungstyps ist **nicht möglich**!

Bei der Ausgabe einer solchen Variablen wird nur der zugehörige Zahlenwert ausgegeben:

```
printf("\n%hu %hu %hu %hu", rot,blau,grau,schwarz);
```

Liefert das Ergebnis `0 1 2 3`

Merke: `%hu` von einen `enum`-Argument wird der Ordinalwert ausgegeben.

Aufzählungstypen

Eine Typumwandlung von einem Auszählungstyp in einen Ganzzahltyp und umgekehrt ist möglich:

```
farbe f = blau;
int i;
farbe f=blau;
i=f-1;
printf("%d\n", i);          0
f=f-2;
printf("%hu\n", f);         unsinnige Ausgabe
printf("%d\n", f);          -1
```

Der zulässige **Wertebereich** wird beim Rechnen mit solchen Variablen **nicht überprüft**.

Beispielprogramm 3-5-enum-test.c

```
 int main(void)
{
  typedef enum {rot, blau, grau, schwarz} farbe;

  farbe f, g, h;
  short x, y;
  int i, j;

  printf("%i",sizeof(farbe));                    4
  printf("\n%hu %hu %hu %hu",
                  rot,blau,grau,schwarz);        0 1 2 3

  f = grau; g = blau; h = g;
  printf("\n%hu %hu %hu",f,g,h);                 2 1 1
  x = (f>g);
  y = (f>schwarz);
  printf("\n%d %d",x,y);                         1 0
```

… 3-5-enum-test.c

```
  i = rot;
  j = g;
  printf("\n%d %d",i,j);

  g = 3;
  h = 4;
  printf("\n%hu %hu",g,h);

  g = f-1;
  h = f+1;
  printf("\n%hu %hu %hu",f,g,h);

  h = f+g;
  printf("\n%hu %hu %hu",f,g,h);
  printf("\n");
}
```

```
0  1

3  4

2  1  3

2  1  3
```

Gültigkeit und Sichtbarkeit von Bezeichnern

In **C89** können Variable **nur am Anfang** eines Anweisungsblockes vereinbart werden:

```
int i;
for (i=0; i < 10; i++) {
      /* do something */
}
```

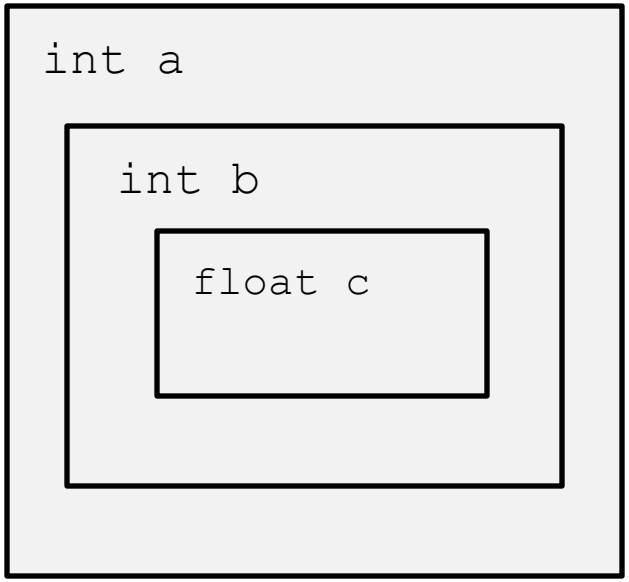
In **C99** bestehen die gleichen Möglichkeiten wie in C++ z.B.

```
for (int i=0; i < 10; i++) {
      /* do something */
}
```

Gültigkeit von Bezeichnern

Der Gültigkeitsbereich (scope) einer Variablen ist der Anweisungsblock, in dem sie vereinbart wurde, einschließlich aller darin enthaltenen inneren Anweisungsblöcke.
Außerhalb dieses Anweisungsblockes ist eine Variable nicht gültig.

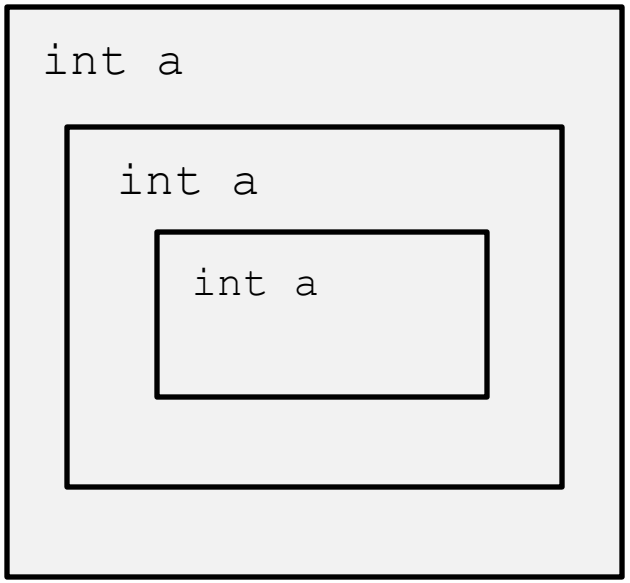
```
{ int a=0;
  printf(„%d\n“, a);
   {
      int b=2;
       printf(„%d  %d\n“, a, b);
       {
          float c=1.1;
           printf(„%d  %d %f\n“, a, b, c);
       }
   }
}
```



Sichtbarkeit von Bezeichnern

Die Sichtbarkeit (visibility) einer Variablen wird eingeschränkt durch die Definition einer Variablen mit dem gleichen Namen in einem inneren Anweisungsblock. Dort ist die weiter außen definierte Variable unsichtbar und es kann nicht auf sie zugegriffen werden, ihr Wert bleibt aber erhalten. Nach Verlassen des inneren Anweisungsblocks kann auf die Variable wieder zugegriffen werden.

```
{ int a=0;
  printf(„%d\n", a);              0
   {
      int a=2;
       printf(„%d\n", a);         2
       {
          int a=1;
          printf(„%d\n", a);      1
       }
   }
}
```



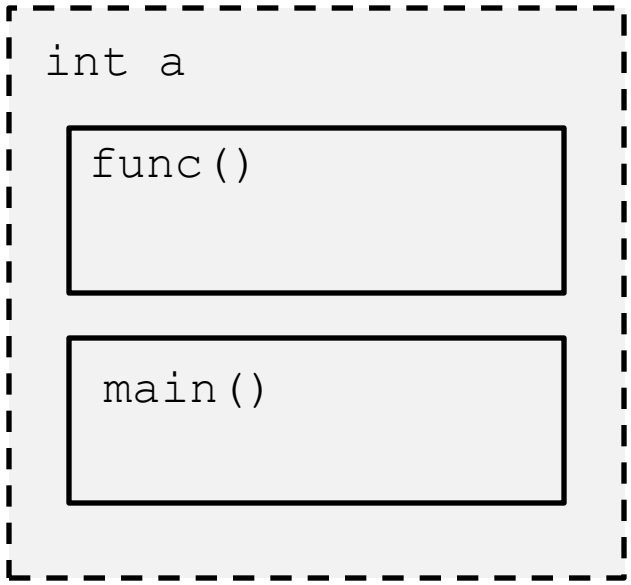
Global Variable

… sind Variable, die außerhalb aller Programmblöcke vereinbart wurden, d.h. globale Variable sind in allen in der Programmdatei enthaltenen Funktionen gültig.

```
int a=10; /* global variable */

void func(){
        a=a+1;
        printf("func: a= %d\n", a);
        return;
}

int main(void){
        printf("main: a= %d\n", a);
        func();
        return 1;
}
```

```
main: a= 10
func: a= 11
```

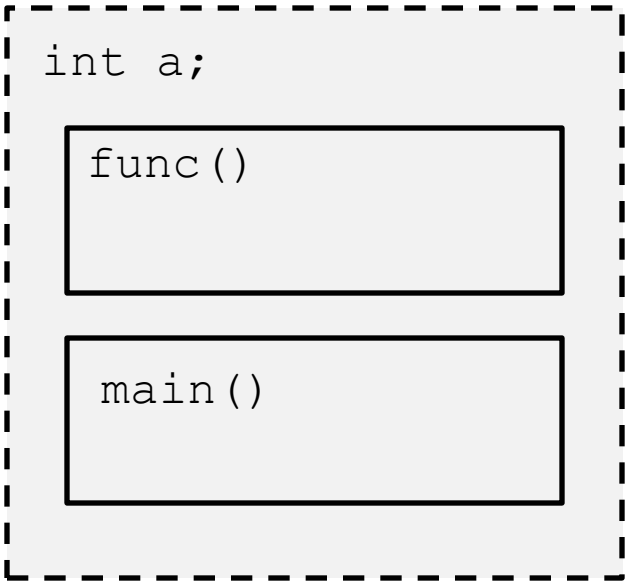
Datei *test-global.c*



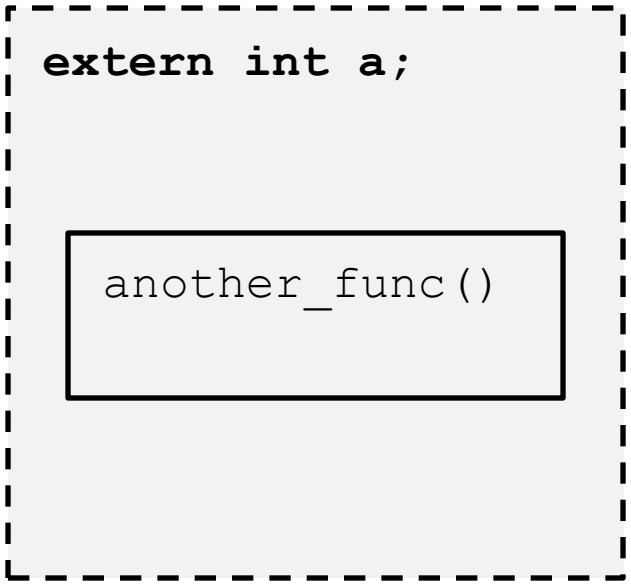
Global Variable

Falls der Programmcode einer Funktion in einer anderen Datei steht, müssen verwendete globale Variable dort als **`extern`** vereinbart werden.

Datei *test-global.c*



Datei *another-func.c*



Global Variable

Wenn allerdings mehrere Funktionen mit den gleichen globalen Variablen arbeiten, wird es sehr **schwierig**:

1. die **Übersicht** zu **behalten**
2. ggf. irgendeinen **Fehler** zu **finden**
3. **Funktionen** in anderem Kontext **wieder** zu **verwenden**

Daher sollte man ***nur dann*** mit globalen Variablen arbeiten, wenn es aus Performance-Gründen unvermeidbar ist!

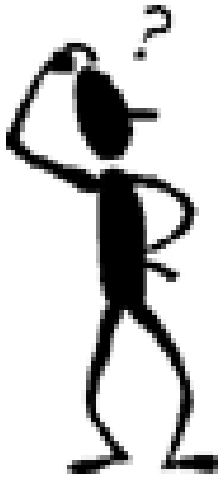
3-6-sichtbar-test.c (Sichtbarkeit von Variablen)

```
 int b=200, d=4;         /* globale Variable */

void btest (void){
      printf("\nbtest: %i",b);
}

int main(void){
      int a=1, b=2;
      printf("\n%i",a);                                    1
      printf("\n%i",b);                                    2
      btest();                                             btest: 200
    /*printf("\n%i",c);    Fehler beim Compilieren! */
      printf("\n%i",d);                                    4
      {
            int b=20, c=30;
            printf("\n\n%i",a);                            1
            printf("\n%i",b);                              20
            btest();                                       btest: 200
            printf("\n%i",c);                              30
            printf("\n%i",d);                              4
      }
      printf("\n\n%i",a);                                  1
      printf("\n%i",b);                                    2
    /*printf("\n%i",c);    Fehler beim Compilieren! */
      printf("\n%i",d);                                    4
}
```

Zwischenbilanz

Die Programmiersprache C stellt eine Reihe „eingebauter“ Datentypen zur Verfügung

- ✓ Ganze Zahlen
- ✓ Gleitkommazahlen
- ✓ Zeichen
- ✓ Aufzählungstypen
- – **Zeiger**



... Fortsetzung folgt!