13.4.4 Konstanten

13.4.4.1 Ganzzahlkonstanten

Ganzzahlkonstanten (Datentyp ***int***) werden nach folgender Syntax geschrieben

Syntax:

dez-integer-konstante : ['+' | '-'] *ziffer-ohne-0* { *ziffer* } .

ziffer-ohne-0 : '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .

ziffer-ohne-0 bezeichnet die dezimalen Ziffer-Symbole von '1' bis '9'

ziffer : '0' | *ziffer-ohne-0* .

ziffer bezeichnet die dezimalen Ziffer-Symbole von '0' bis '9'.

Beginnt die Zahlenfolge mit **0x**, so wird sie hexadezimal interpretiert. In diesem Fall zählen auch A,B...F (bzw. a,b,...f) zu den erlaubten Zeichen.

Beispiele:

```
78,  +675,  -10,   0xaf33,   0x123F
```

(Nur zur Info: mit führender Null: Interpretation als Oktalzahl: 0100 = 64 (!))

13.4.4.1 Ganzzahlkonstanten (Fortsetzung)

		Wertebereich
int (=signed int**)** **unsigned int**	z.B. 4 Byte (s.u.) "	-2^{31} .... $2^{31}-1$ (-2147483648 .... 2147483647) 0 .... $2^{32}-1$ (0 .... 4294967295)
long int **unsigned long int**	meist 4 Byte "	-2^{31} .... $2^{31}-1$ (-2147483648 .... 2147483647) 0 .... $2^{32}-1$ (0 .... 4294967295)
short int **unsigned short int**	meist 2 Byte "	-2^{15} .... $2^{15}-1$ (-32768 .... 32767) 0 .... $2^{16}-1$ (0 .... 65535)
char (=signed char**)** **unsigned char**	1 Byte "	-2^{7} .... $2^{7}-1$ (-128 .... 127) 0 .... $2^{8}-1$ (0 .... 255)
[long long int	meist 8 Byte	-2^{63} .... $2^{63}-1$]

Laut ANSI muss die ***int***-Variable mindestens eine Größe von 16 Bit aufweisen
Größe von **int** ist maschinenabhängig:

- auf 16-bit-Maschinen (z.B. 68000) ist **int** i.Allg. 16 Bit groß,
- auf 32-bit-Maschinen (z.B. PowerPC, ARM) ist **int** i.Allg 32 Bit groß.

Hintergrund: um C als Implementierungssprache auf den verschiedensten Rechnersystemen (Mikrocontroller, Großrechner) einsetzen zu können.

Anm.: die gültigen Größen stehen in "*limits.h*"

Beispiel: Auszug aus `limits.h` (GNU ARM)

```
....
....

#define SHRT_MIN   (-32767-1)        /* minimum signed   short value */
#define SHRT_MAX     32767           /* maximum signed   short value */
#define USHRT_MAX    65535U          /* maximum unsigned short value */

#define LONG_MIN   (-2147483647L-1)  /* minimum signed   long value */
#define LONG_MAX     2147483647L     /* maximum signed   long value */
#define ULONG_MAX    4294967295UL    /* maximum unsigned long value */

#define INT_MIN      LONG_MIN        /* minimum signed   int value */
#define INT_MAX      LONG_MAX        /* maximum signed   int value */
#define UINT_MAX     ULONG_MAX       /* maximum unsigned int value */
....
....
```

13.4.4.2 Gleitkommakonstanten

Gleitkommakonstanten (Datentyp ***float***) gehorchen der folgenden Syntax:

Syntax:

gleitpunkt-konstante : ['+' | '-'] (*dez-darstellung* | *exp-darstellung*) .

dez-darstellung : *ziffer { ziffer }* '.' *{ ziffer }* | *{ ziffer }* '.' *ziffer { ziffer }* .

exp-darstellung : (*ziffer { ziffer }* | *dez-darstellung*) ('e' | 'E') ['+' | '-'] *ziffer { ziffer }*

(Exponent zur Basis 10: 1e3 = 1000)

Unter Ziffer wird die Menge der dezimalen Ziffer-Symbole von '0' bis '9' verstanden.

Beispiele:

```
78.4,  +0.675,  -1045.125576,  5. ,  .45
0.784e2,   +67.5E-2,   -0.1045125576e+4,  3e5
```

13.4.4.2 Gleitkommakonstanten (Fortsetzung)

		Wertebereich	signifikante Dezimalstellen
float	meist 4 Byte	$3.4*10^{-38}$ ..... $3.4*10^{38}$	7
double	meist 8 Byte	$1.7*10^{-308}$ ..... $1.7*10^{308}$	16
long double	meist 10 Byte	$3.4*10^{-4932}$ ..... $3.4*10^{4932}$	19

Anm.:

- Die gültigen Größen stehen in "*float.h*".
- Viele Maschinen besitzen eine 64 Bit Floatingpoint-Einheit, so dass der Datentyp **float** keine Rechenzeitvorteile bringt.
  Der Datentyp **double** ist daher i.allg. die bessere Wahl.
- **long double** ist vor allem dann zweckmäßig, wenn eine hohe numerische Genauigkeit erforderlich ist (z.B. sog. schlecht konditionierte Gleichungs-systeme, nummerische Methoden).

Beispiel: Auszug aus `float.h` (Borland C++-Builder)

```
......
......
#define   DBL_MANT_DIG          53
#define   FLT_MANT_DIG          24
#define   LDBL_MANT_DIG         64

#define   DBL_EPSILON           2.2204460492503131E-16
#define   FLT_EPSILON           1.19209290E-07F
#define   LDBL_EPSILON          1.08420217248550443 4e-019L

/* smallest positive IEEE normal numbers */
#define   DBL_MIN               2.2250738585072014E-308
#define   FLT_MIN               1.17549435E-38F

#define   DBL_MAX_10_EXP       +308
#define   FLT_MAX_10_EXP       +38
#define   LDBL_MAX_10_EXP      +4932
.....
```

13.4.4.3 Zeichenkonstanten

Zeichenkonstanten (Datentyp ***char***) =
einzelne in einfache Anführungsstriche eingeschlossene Zeichen:

Beispiele: `'a', 'R', '1', '8'`

Der Wert der Konstante ist der numerische Wert des Zeichens im Zeichensatz der jeweiligen Maschine.

Darüberhinaus sind sog. Escape-Sequenzen erlaubt, z.B. (s. ASCII-Tabelle):

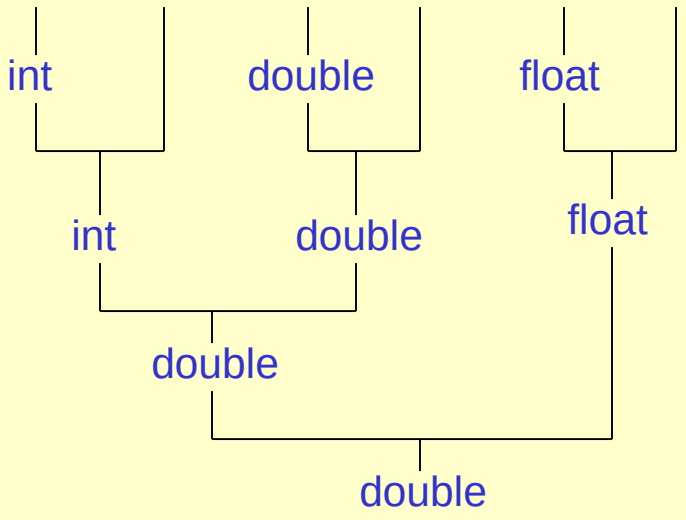
`\n`	Zeilenumbruch (CR)
`\r`	Wagenrücklauf (LF)
`\f`	Seitenwechsel (FF)
`\b`	Backspace (BS)
`\0`	Nullzeichen (NUL)
...	...

13.4.4.4 Implizite Typkonvertierungen

Werden in math. Ausdrücken verschiedene Typen kombiniert, wird bei jeder Operation der "niedrigere" Typ in den “höheren“ konvertiert.

Beispiel:

```
char   ch;
int    i;
float  f;
double d;
...
result = (ch / i) + (f * d) - (i + f);
```



13.4.4.5 Explizite Typkonvertierungen (= casting)

Gelegentlich werden auch explizite Typvereinbarungen benötigt. Hierzu wird im math. Ausdruck vor die zu konvertierenden Variablen der Zieltyp in Klammen gesetzt:

Beispiel:

```
int    a=7;
int    b=2, c;

double Erg1, Erg2;

...

c = a/b;  // Ergebnis = ?

Erg1 = (double)a/(double)b;  /* Erg1 = 3.5     */

Erg2 = a/b;                  /* Erg2 = 3.0 !!! */
```

13.4.5 Strings

String (Zeichenkette = Folge von Zeichen umgeben von Anführungszeichen ("...") .

Intern wird der Zeichenkette ein NUL-Zeichen (\0) angehängt, wodurch das Ende der Zeichenkette markiert wird.

Beispiel: `"ABC 123 \n"`

Im Speicher steht dann:

0x41	*0x42*	*0x43*	*0x20*	*0x31*	*0x32*	*0x33*	*0x20*	*0x0D*	*0x00*

ASCII-Merkregel:

Zahlen beginnen bei 0x30.
Großbuchstaben beginnen bei 0x41.
Kleinbuchstaben beginnen bei 0x61.
Leerzeichen ist 0x20.

13.4.6 Operatoren

13.4.6.1 Binäre Operatoren: Übersicht

+	Addition	Arithmetik	Zahlen (u. teilw. Zeiger)
-	Subtraktion		"
*	Multiplikation		"
/	Division		"
%	Modulo-Division (Rest)		ganze Zahlen
<	kleiner	Vergleich	alle Typen
<=	kleiner gleich		"
==	gleich		"
!=	nicht gleich		"
>=	größer gleich		"
>	größer		"
&	bitweise AND	Bitoperationen	ganze Zahlen
|	bitweise OR		"
^	bitweise XOR		"
<<	bitw. linksschieben		"
>>	bitw. rechtsschieben		"
&&	log. AND	Logik	boolsche Werte
||	log. OR		"

13.4.6.2 Binäre Operatoren: Arithmetische Operatoren

Ausdrücke mit arithmetischen Operatoren liefern einen numerischen Wert.

Bei der Auswertung eines Ausdruckes wird der Vorrang der Operatoren beachtet.

- 3 + 5 * 4 – 2 liefert den Wert 21, weil der Vorrang von * beachtet wird.
- (3 + 5) * (4 – 2) liefert den Wert 16, weil durch die Klammern die Addition und die Subtraktion Vorrang vor der Multiplikation erhalten.

Vorrangstufen der numerischen Operatoren

Stufe	Operanden	Erläuterung	Auswertung
5	( )		von links
4	+ -	unären Operatoren (Vorz.)	von rechts
3	* % /		von links
2	+ -	binären Operatoren	von links
1	=	Zuweisung	von rechts

13.4.6.2 Binäre Operatoren: Anmerkung zur Auswertereihenfolge

Stehen mehrere (bezüglich der Vorrangstufe) gleichwertige Operationen hintereinander, sind die Reihenfolgeregeln anzuwenden:

Beispiel: Die ganzzahlige Berechnung von "3 * 11 / 4" ergibt:

a) von links ausgewertet (3 * 11) / 4 = 33 / 4 = 8 !
b) von rechts ausgewertet 3 * (11 / 4) = 3 * 2 = 6 !

--> Das Assoziationsgesetz gilt nicht mehr !!!!
C würde Lösung a) liefern.

Zur Vermeidung solcher schwer durchschaubarer Rechenregeln sollte man entweder

a) Klammern : Klammern machen zweifelsfrei die Reihenfolge deutlich
b) oder einen höheren Zahlentyp wählen (z.B. double).
(in diesem Fall wäre das Ergebnis 8.25)

ÜBUNG: Arithm. Operatoren

Gegeben seien:

```
char    c1='5', c2=25, c3, c4=-1;
int     i=5, j=9, k=-15, m, n, p, q;
double  d1=12.5, d2=2.0E-3, d3=-100, d4, d5, d6;
```

Berechnen Sie

```
m  = d1*i;
d4 = 9/2;
n  = k%4;
j  = j+1;
p  = d2*750 + 0.1;
d5 = (double)j/i + 0.2;
q  = (unsigned char)c4;
d6 = 22%5*3%2;
c3 = (c1-0x30)+c2;
c3 = 64*8;
```

Welche Ausdrücke zeugen von schlechtem Stil ?

13.4.6.3 Binäre Operatoren: Vergleichsoperatoren

Vergleichsoperatoren liefern ebenfalls ein numerisches Ergebnis:

- Vergleich falsch: 0
- Vergleich richtig: 1

Bei der Auswertung ist auf den Vorrang zu achten.

Vorrangstufen der numerischen Operatoren

Stufe	Operanden	Erläuterung	Auswertung
7	( )		von links
6	+ -	unären Operatoren	von rechts
5	* % /		von links
4	+ -	binären Operatoren	von links
3	< <= => >		von links
2	== !=		von links
1	=	Zuweisung	von rechts

Beispiele

3 < 5 - 4	liefert den Wert 0	
(3 < 5) - 4	liefert den Wert - 3	(Unsinn: Typ-Kuddelmuddel)
3 < 5 < 4 < 2	liefert den Wert 1	(Unsinn: “ “)

13.4.6.4 Binäre Operatoren: Logische Operatoren

Logische Operatoren (&&, ||, !) liefern ebenfalls ein numerisches Ergebnis:

- Aussage falsch (false): 0
- Aussage richtig (true): 1

Die logischen Operatoren stehen in engem Zusammenhang mit den Vergleichsoperatoren.

Bei der Auswertung ist auf den Vorrang zu achten .

Vorrangstufen der log. Operatoren und Vergleichsoperatoren

Stufe	Operanden	Erläuterung	Auswertung
7	!		
6	< <= => >		von links
5	==, !=		von links
4	&&	log. Operatoren (AND)	
3	||	" " (OR)	
1	=	Zuweisung	von rechts

In C wird jeder Ausdruck **ungleich 0** logisch interpretiert als wahr (true) betrachtet !!

ÜBUNG Vergleichsoperatoren, log. Operatoren

Was ergeben folgende Ausdrücke:

```
char  b1, b2, b3, b4, b5, b6, b7;
int   Zahl=7, Z2=25;

b1 = !(Zahl > 10) && !(Zahl < 5);
b2 = (Zahl <=10) && (Zahl >= 5);
b3 = Zahl <= 10 && Zahl >= 5;
b4 = (Zahl - 10) && (Zahl - 7);
b5 = 5 <= Z2 <= 10;
b6 = 5 <= Z2 && Z2 <= 10;
b7 = !(Zahl=5);
b8 = !(Z2 < 30) || Z2==25 || !(Z2-25);
```

Welche Ausdrücke zeugen von schlechtem Stil ? Welche sind unsinnig ?

ÜBUNG Schaltjahrberechnung

Zu einer gegebenen Jahreszahl ist zu berechnen, ob es sich um ein Schaltjahr handelt. Die Regel für Schaltjahre lautet:

Ein Schaltjahr liegt dann vor, wenn

- die Jahreszahl durch 4 teilbar ist,
- außer sie ist durch 100 teilbar.
- Einzige Ausnahme: Ist die Jahreszahl durch 400 teilbar, liegt jedoch trotzdem ein Schaltjahr vor.

Sind die Jahre 1800 und 2000 Schaltjahre ?

Geben Sie einen Ausdruck an, der den log Wert 1 (true) ausgibt, wenn ein Schaltjahr vorliegt.

13.4.6.5 Unäre Operatoren: Übersicht

& *	Adresse von ... Inhalt von ...	Referenzierung Dereferenzierung	alle Typen Zeiger
+ -	pos. Vorzeichen neg. Vorzeichen	Arithmetik "	Zahlen "
~	bitweise invertieren	Bitoperation	ganze Zahlen
!	log. invertieren	Logik	boolesche Werte
(Zieltyp)	Typumwandlung		
sizeof	Speicherbedarf		Ausdrücke u. Typen
++ --	Incrementierung Decrementierung	Prä- und Post-increment/-decrement	ganze Zahlen und Zeiger

13.5 Variablen und Vereinbarungen

Variablen = Datenobjekte, deren Wert im Programmverlauf geändert werden kann.
Alle Variablen müssen vor ihrer ersten Benutzung in einem Programm deklariert werden.

Datenobjekte werden nach folgender Syntax vereinbart.

Syntax:

declaration : *type-specifier* *init-declarator* { ',' *init-declarator* } ';' .

type-specifier : 'int' | 'float' | 'double'|.....usw... . Anm.: unvollständig !

init-declarator : *variablen-bezeichner* ['=' *initialwert*] .

Beispiele:

```
int     zaehler, increment = 5 ;
float   pi = 3.1415926 ;
```

vorgezogener Teil 13.13-13.19: Adressen, Felder, Strings

13.13 Adressen und Zeiger

Zeiger

- ***eine Variable, die eine Adresse enthält***
- ***ist Typ-gebunden ("zeigt auf Variable des Typs …..")***

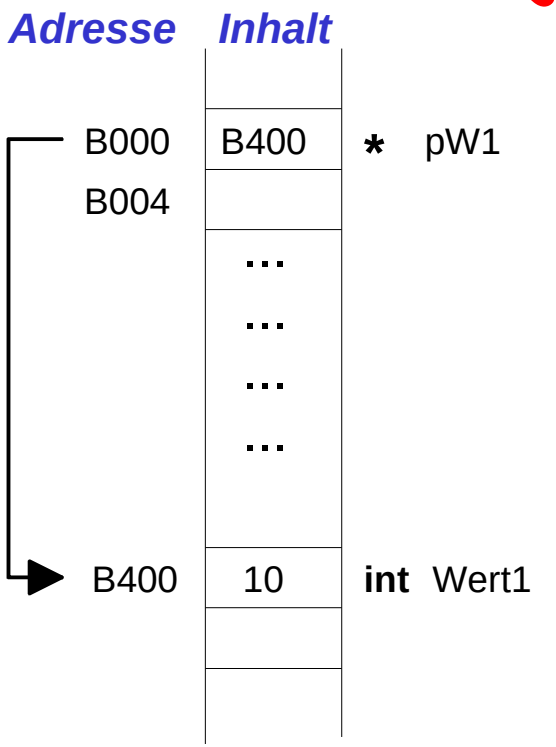
Deklaration:

- ***durch * vor Variablenname***

Initialisierung:

- ***Zeiger muss mit gültiger Adresse belegt werden***
- ***Die Adresse einer Variablen erhält man durch den Adressoperator &.***

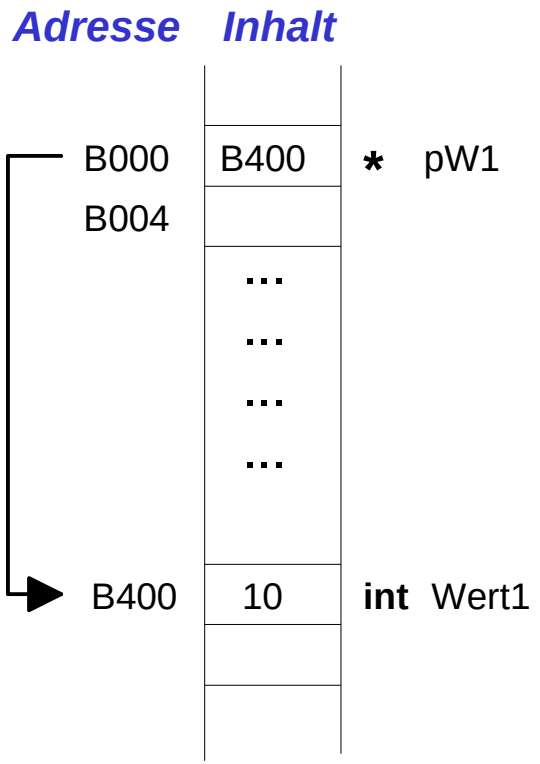
```
/* Deklaration */
int  Wert1 = 10;
int  *pW1;  // pW1 ist Zeiger auf Integer

/* Initialisierung */
pW1 = &Wert1;
```



13.14 Zugriff auf Daten über den Zeiger (*Dereferenzierung*)

Zugriff auf den Zeigerwert (= Wert von derjenigen Variablen holen, auf die der Zeiger zeigt)
---> durch * -Operator vor dem Zeigernamen

```
/* Deklaration */
int  Wert1 = 10,  *pW1;

/* Initialisierung */
pW1 = &Wert1;

/* Zugriff */
printf("Wert %d, Adresse %X", *pW1, pW1);
```



WICHTIG:

In der Deklaration bedeutet *pWert1: "pWert1 ist ein Zeiger auf Typ ....."

Im Programmtext bedeutet *pWert1: „Der Wert, auf den der Zeiger pWert1 zeigt, ist ….. (→ Dereferenzierung)

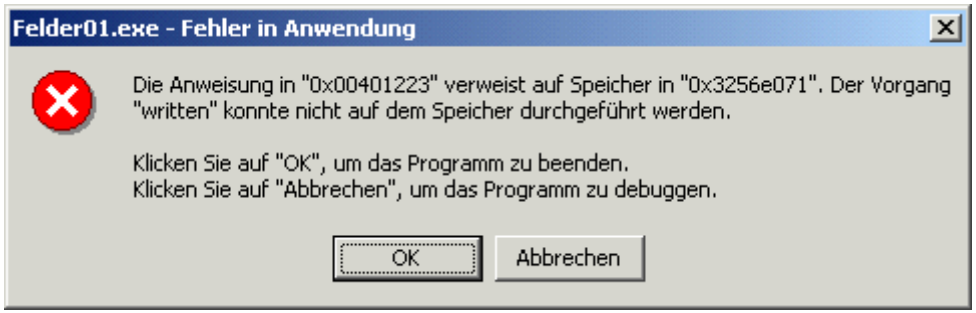
13.14 Zugriff auf Daten über den Zeiger (Fortsetzung)

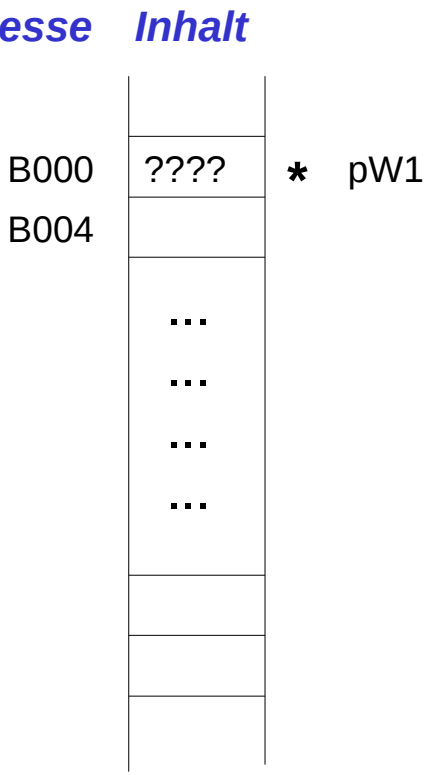
Achtung: Fehlergefahr

--> "dangling pointer"

```
/* Deklaration */
int  *pW1;

/* Zugriff */
*pW1=10; // Der Wert auf den der
         // Zeiger zeigt soll 10 sein.
```

..... aber wohin zeigt der Zeiger ?

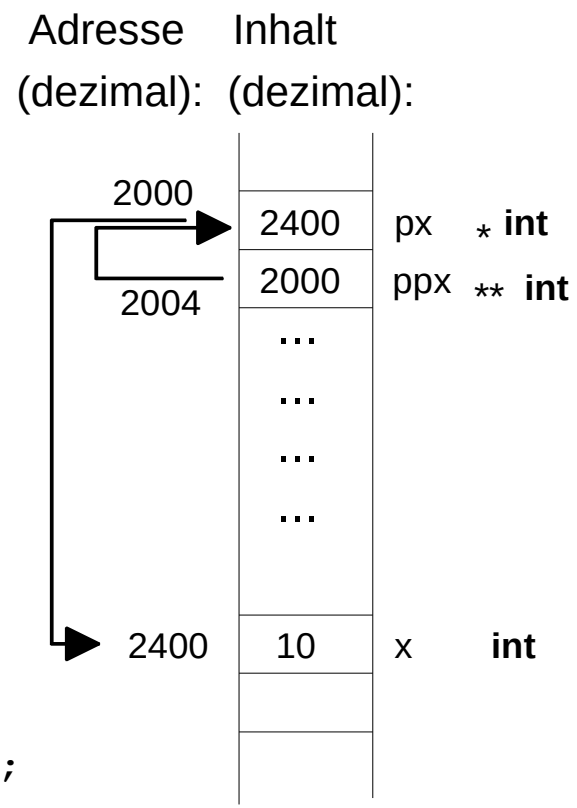




ÜBUNG: Einfache Zeigeroperationen

Gegeben ist folgendes Programm:

```
int x = 10;
int *px;
int **ppx;

px  = &x;
ppx = &px;

printf("%d %d %d", x,    &x,     px);
printf("%d %d %d", &px,  *px,    *ppx);
printf("%d %d %d", &ppx, **ppx,  ppx);
```



Gegeben sei auch nebenstehendes Speicherbild (Memorymap).
Was wird ausgegeben ?

13.15 Ein- und mehrdimensionale Felder

13.15.1 Definition

Feld: Menge von gleichartigen Variablen, die durch einen einzelnen Namen repräsentiert werden

Definition: *Typ Variablenname [Anzahl der Elemente]*

Zugriff: Index des ersten Feldes ist 0 !

```
/* Definition */
int x[2];

/* Initialisierung */
x[0]=12;
x[1]=25;

printf('x0 ist %d \n', x[0]);
```

```
/* Definition */
char txt[2];

/* Initialisierung */
txt[0]='A';
txt[1]='b';

printf('x0 ist %c \n', txt[0]);
```

Fazit: ***Es gibt keinen Unterschied zwischen Feldern mit Zahlen und Zeichen ! Zeichen werden auch als Zahlen (ASCII) abgespeichert.***

13.15.2 Definition und Initialisierung

```
/* mit Angabe der Feldgrösse */
int  x[10]  = {3, 6, 9, 12, 15, 18, 21, 24, 27, 30};
char txt[5] = {'a', 'b', 'c', 'd', 'e' };


/* ohne Angabe der Feldgrösse */
int  x[]   = {3, 6, 9};
char txt[] = {'a', 'b', 'c', 'd', 'e' };


/* Teilinitialisierung */
int  x[10]  = {3, 6};
char txt[5] = {'a', 'b'};
```

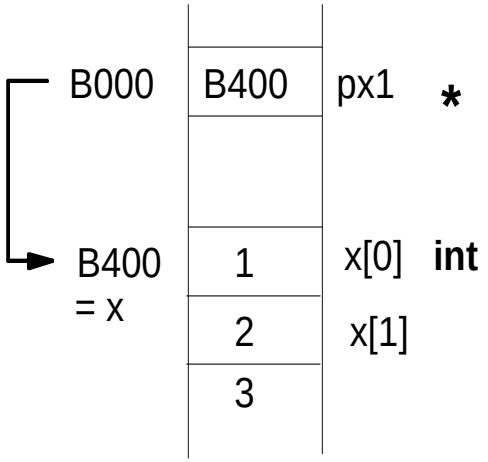
13.15.3 Mehrdimensionale Felder

```
/* mit Angabe der Feldgrösse */
int x2[2][3];     /* 6 Werte in 2 Feldern und je 3 Elementen  */


char c2[3][2] = {{'t', 'f'},  /* Feld von 3 Elementen (3 Zeilen)  */
                 {'f', 'f'},  /* mit jew. 2 Elementen (2 Spalten) */
                 {'f', 'x'}};


char c2[][2]  = {{'t', 'f'},  /* Felddimension 1  (nur diese)  */
                 {'f', 'f'},  /* kann entfallen.               */
                 {'f', 'x'}};


int x3[4][3][2] =  {  { {0,1},{1,0},{1,1} },
                      { {1,1},{0,0},{1,0} },
                      { {0,0},{1,1},{1,1} },
                      { {0,1},{0,1},{0,1} }   };
```

Achtung: Die geschweiften Klammern werden vom Compiler nicht ausgewertet !

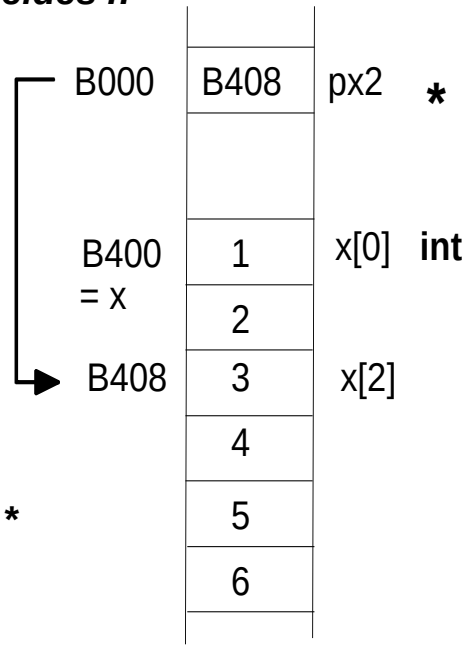
13.16 Felder und Zeiger

```
int x[]  = {1, 2, 3, 4, 5, 6, 7};
int *px1 = x;          /* = &x[0] */

/* Folgende Schreibweisen sind äquivalent */
printf("Adr. des 1.Zeichens=%X", &x[0]);
printf("Adr. des 1.Zeichens=%X", x);
printf("Adr. des 1.Zeichens=%X", px1);
```



Wichtig: ***Der Feldname ohne Index liefert die Adresse des 1. Feldes !!***

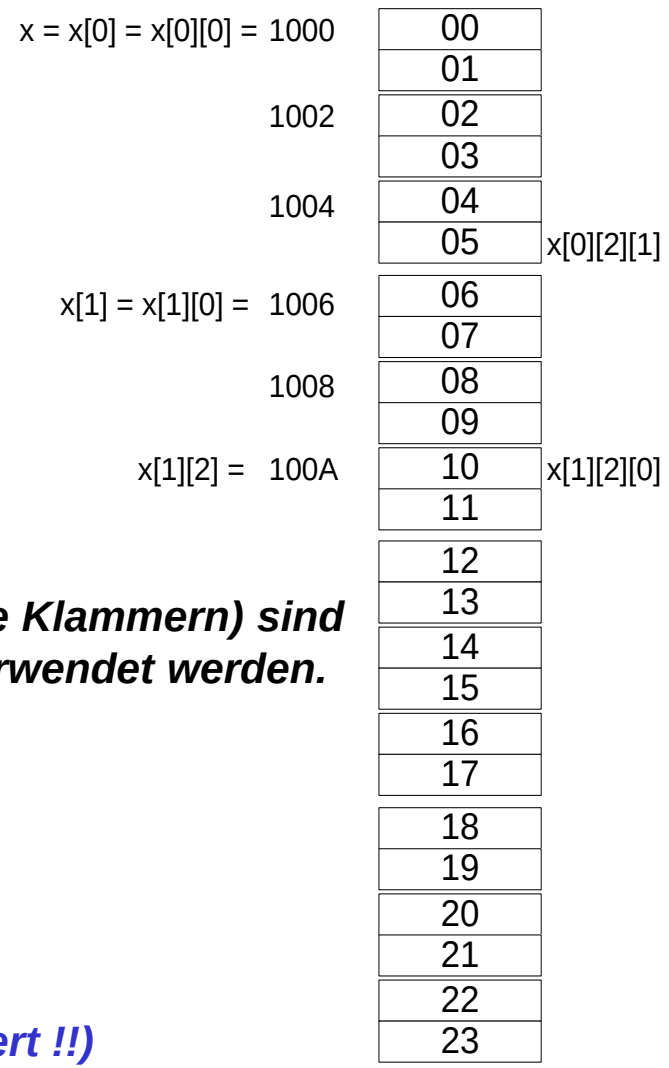
```
int x[]  = {1, 2, 3, 4, 5, 6, 7};
int *px2 = &x[2];


/* Ausgabe :   3  3  6 */
printf("%d  %d  %d", *px2,  px2[0],  px2[3]);
```



Wichtig: ***Ein indizierter Zeiger liefert den indizierten Wert !!***
= Dereferenzierung von Zeigern durch [] statt durch *

13.16 Felder und Zeiger (Fortsetzung)

```
/* Definition und Initialisierung */
 char x[4][3][2] =
      { { { 0, 1},{ 2, 3},{ 4, 5} },
        { { 6, 7},{ 8, 9},{10,11} },
        { {12,13},{14,15},{16,17} },
        { {18,19},{20,21},{22,23} } };

/* Zugriff auf Feldelemente */
a=3; b=2; c=1;
y = x[a][b][c];      /* y= 23*/
```



Wichtig: ***<u>Unvollständige Feldnamen</u> (fehlende Klammern) sind Adressen und können <u>wie Zeiger</u> verwendet werden.***

Beispiele:

- ***x = x[0] = x[0][0] = 1000***
- ***x[1] = x[1][0] = 1006***
- ***x[1][2] = 100A***
- ***x[1][2][0] = 10 (der Wert !!)***

ÜBUNG: Umgang mit Zeigern

Zeichnen Sie die Memorymap. Was wird ausgegeben ?

```
int a[]={1,2,3,4,5,6,7};

int main(){

        int *p1;
        int **p2;

        p1=&a[2];
        p2=&p1;

        printf("%d\n",a[3]);
        printf("%d\n",*p1);
        printf("%d\n",p1);
        printf("%d\n",&p1);
        printf("%d\n",p2);
        printf("%d\n",&p2);
        printf("%d\n",p2[0][2]);

        return 0;
}
```

Folgende Annahmen gelten:

Das Feld a beginne bei Adresse 1000 (dezimal).

Der Zeiger p1 stehe bei Adresse 2000 (dez.) gefolgt vom Zeiger p2.

Achtung:

Das Beispiel hat didaktischen Wert, zeugt aber nicht gerade von gutem Stil !

13.17 Zeichenketten (Strings)

13.17.1 Definition und Initialisierung

String

- Sequenz von Zeichen (-> Feld),
- in Anführungszeichen eingeschlossen,
- mit '\0' abgeschlossen ist (Null-Zeichen).

```
char txt1[]  = "AB1";
char txt2[4] = "AB1"; /* Platz für Nullzeichen nicht vergessen */
char txt3[]  = {'A','B','1','\0'};

/* String ausgeben mit %s und Stringname */
printf("%s %s %s", txt1, txt2, txt3);
```

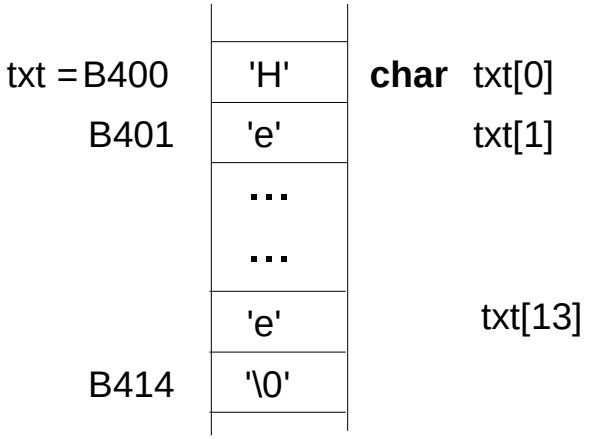
= 0x41 0x42 0x31 0x0

13.18 Strings und Zeiger

Feldnamen können wie Zeiger verwendet werden,
(sind nur Adressen) !

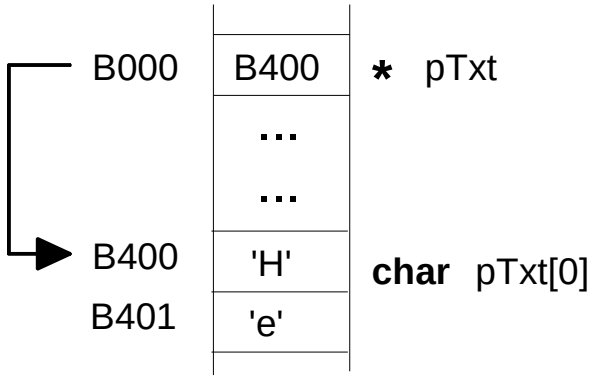
```
char txt[] = "Herr der Ringe";

printf("Adr. des 1.Zeichens=%X", txt);
```

= Adresse des 1 . Zeichens




```
char *pTxt = "Herr der Ringe";

printf("Adr. des 1.Zeichens=%X", pTxt);
```

Adresse auf die der Zeiger zeigt
= Adresse des 1 . Zeichens



ÜBUNG: Beispiele zu Feldern und Strings

1. Schreiben Sie ein C-Programm, welches eine 4x3-Matrix mit einem Vektor multipliziert.

2. Schreiben Sie ein C-Programm, welches die Grossbuchstaben eines Strings zählt und ausgibt.

13.19 Funktionen mit Call-by-reference-Parameterübergabe

```
/* Funktionsdeklaration */
void SetString2x(char *str);



/* Aufrufendes Programm */
int main () {

    char txt[] = "Autobahn";

    /* Funktionsaufruf */
    SetString2x(txt);
    printf("%s", txt);

    return 0;
}
```

```
void SetString2x(char *str){

   int i=0;

   while(str[i] != '\0'){
        str[i]='x';
        i++;
   }
}
```

Wichtig: ***Mit Hilfe von Zeigern werden Adressen an Unterprogramme übergeben.***

Anm.:

`str[i]` ist äquivalent zu `*(str+i)`

ÜBUNG: Call-by-reference, Umgang mit Matrizen

Schreiben Sie ein Unterprogramm, welches ein Feld von Intergerwerten der Größe nach sortiert.

Ansatz: Durchlaufe das Feld immer wieder und vertausche jedesmal, wenn nötig, benachbarte Elemente.

Der Vorgang kann abgebrochen werden, wenn bei einem Durchlauf kein Vertauschen mehr vorkommt (--> Bubblesort).

Weiter ist das Hauptprogramm zu schreiben, welches das Unterprogramm aufruft.

ÜBUNG: Call-by-reference, Umgang mit Strings

Schreiben Sie ein Unterprogramm, welches zwei Strings lexikographisch vergleicht.

Ansatz: Die beiden Strings (s, t) werden Zeichen für Zeichen miteinander verglichen. Sobald ein Unterschied festgestellt wird, wird der Vorgang abgebrochen.

In diesem Fall wird die Differenz der ASCII-Werte der zuletzt betrachteten Zeichen ausgegeben.

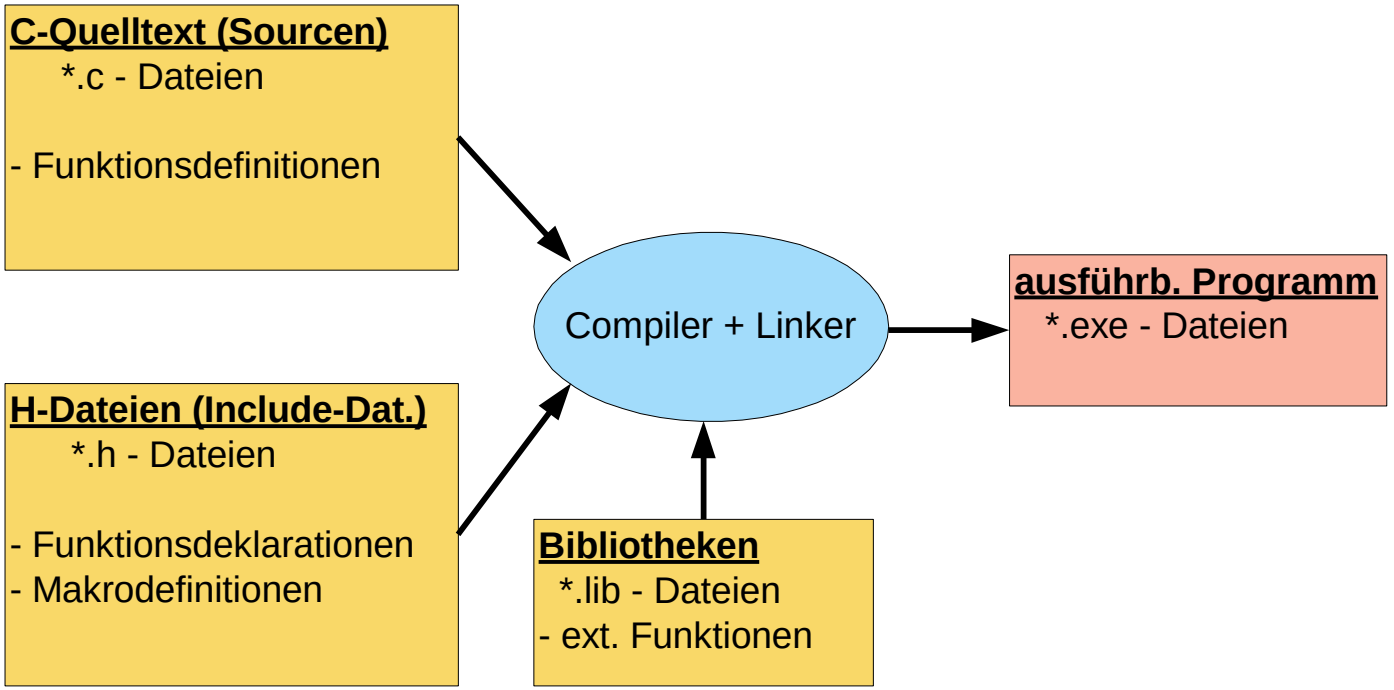
Sind die beiden Strings gleich, wird 0 zurückgegeben.

Weiter ist das Hauptprogramm zu schreiben, welches das Unterprogramm aufruft.

Ende vorgezogener Teil 13.13-13.19

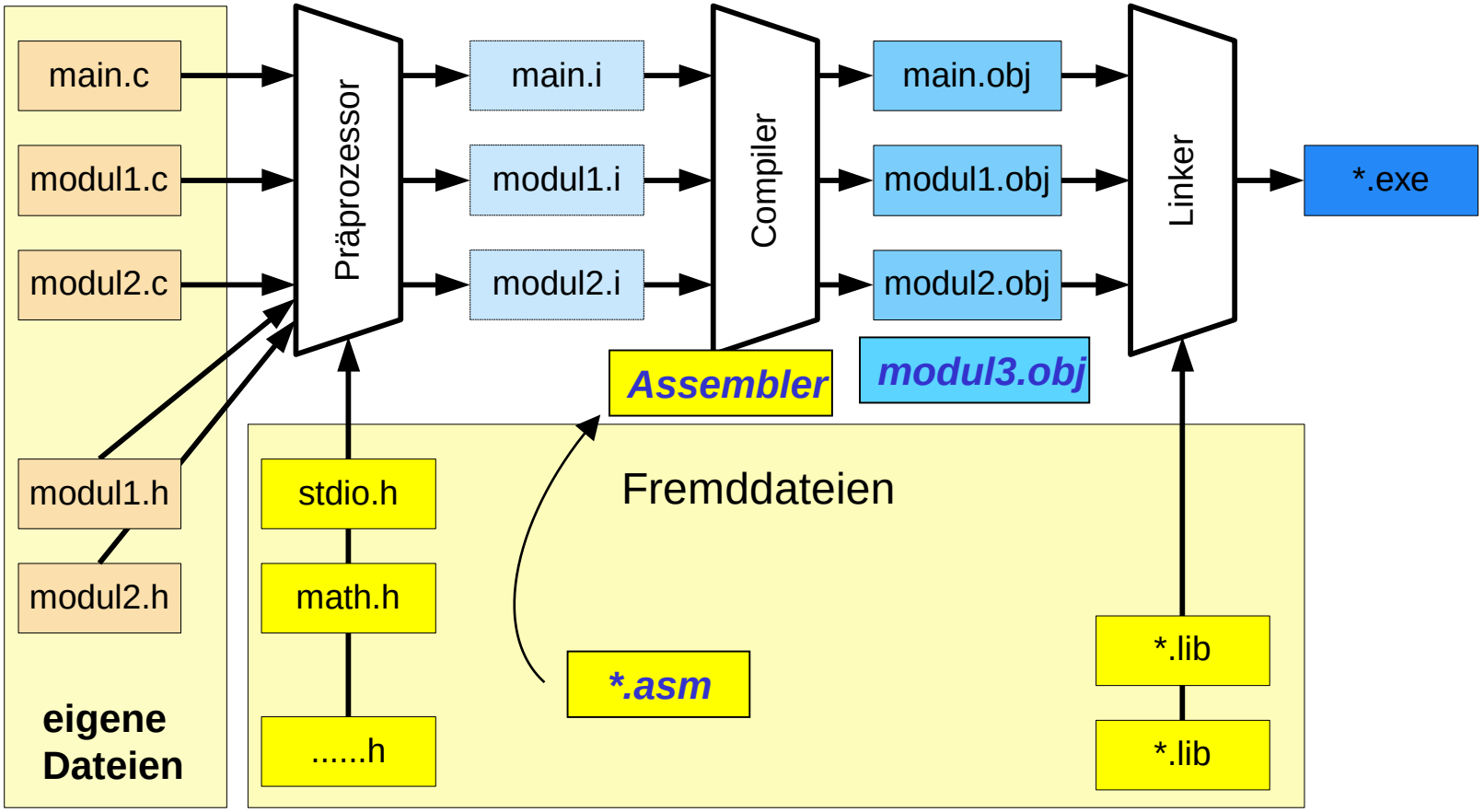
13.24 Softwareerstellung unter C

13.24.1 Grundlegende Überlegungen



Ziel: Erzeugung eines ausführbaren Programms für das Zielsystem aus dem Sourcecode.

13.24.2 Schritte zur Erzeugung eines ausführbaren Programms



Erinnerung Parameter-übergabe

10.2.2 Parameterübergabe über den Stack

10.2.2.1 Einleitende Anmerkungen

Eine übliche Konvention beim ARM Thumb-Befehlssatz ist, die ersten vier Parameter über r0 – r3 zu übergeben und alle weiteren Parameter über den Stack.

Dies ermöglicht für die meisten Unterprogramme (0 ... 4 Parameter) einen sehr schnellen Unterprogrammsprung (mit wenig Overhead), ermöglicht aber auch mehr (5 ... beliebig viele) Übergabeparameter (mit etwas mehr Overhead).

Bei vielen anderen Prozessoren (mit weniger Registern) ist die Parameterübergabe über den Stack der Standard.

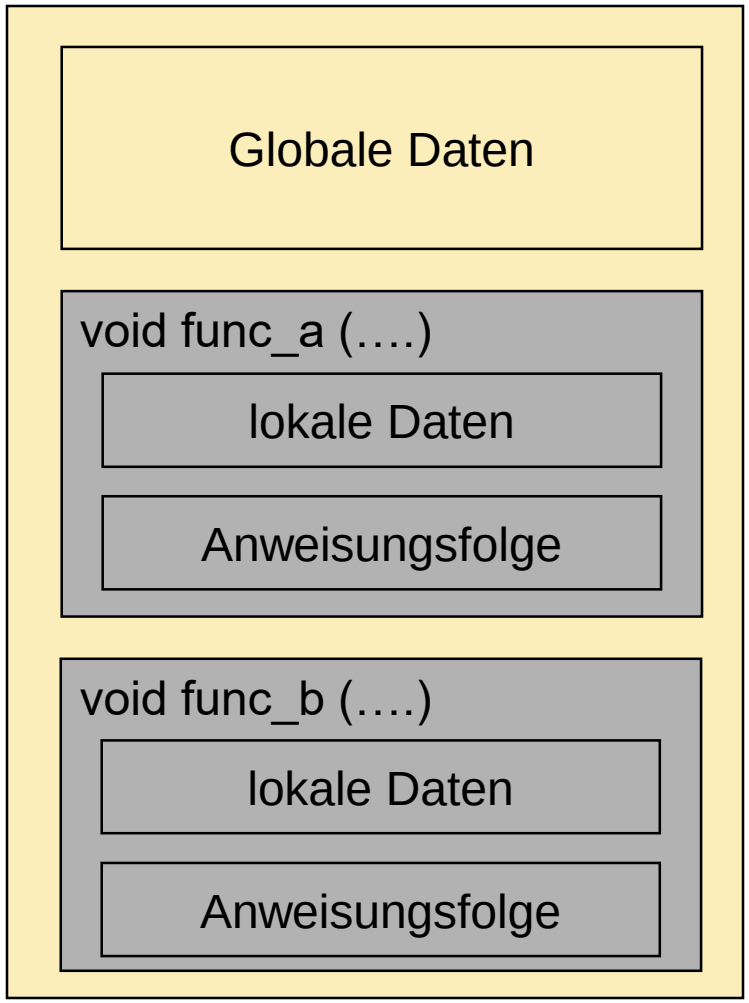
Die Parameter r4 - r11 stehen üblicherweise als lokale Variablen zur Verfügung.

Werden mehr lok. Variablen benötigt, so kann auch hierfür der Stack verwendet werden (s.u.).

Die Parameterübergabe nach Konvention ist dann einzuhalten, wenn die Assembler-Unterprogramme auch von C-Programmen aufgerufen werden sollen.

vorgezogenes Kapitel 13.9

13.9 Gültigkeitsbereiche und Speicherklassen



Dargestellt ist ein Modul (Texteinheit), z.B. die Datei „*irgendeinName.c*“.

Die globalen Daten sind in allen Funktionen bekannt und die lokalen Daten nur in ihren Funktionen.

Die globalen Daten sind für die Funktionen extern.

Die lokalen Daten haben die gleiche Lebensdauer wie ihre Funktionen.

Wenn die Funktionen aufgerufen werden, werden die Datenobjekte angelegt. Sie werden aufgelöst, wenn die Funktion endet.

In diesem Sinn unterscheidet C zwischen statischen und automatischen Datenobjekten.

13.9 Gültigkeitsbereiche und Speicherklassen (Fortsetzung)

Es gibt vier Speicherklassen:

auto Lokale Variablen einer Funktion werden erzeugt wenn die Funktion aufgerufen wird und gelöscht (*), wenn die Funktion verlassen wird. Solche Variablen heißen "automatische Variablen".

static Innerhalb einer Funktion deklariert: Lokale Variablen, die ihren Wert zwischen den Funktionsaufrufen behalten.
Außerhalb der Funktionen deklariert: Modul-globale Variablen, d.h. innerhalb des Moduls global, außerhalb des Moduls unbekannt.

extern Außerhalb der Funktionen deklarierte Variablen stehen allen Funktionen des Moduls zur Verfügung (= globale Variablen).
Durch Deklaration mit "extern" können diese Variablen sogar über Modulgrenzen hinweg *importiert* werden.

register Variablen, die möglichst in den Prozessorregistern gehalten werden sollten (Geschwindigkeitsvorteil).

(*) gelöscht werden die Variablen/Verwaltung, nicht die Daten im Speicher selbst

ÜBUNG: (Gültigkeitsbereiche)

```
/* --- Modul_1.c --- */
#include "Modul_2.h"

int calc_i();     /* Fkt.-Deklaration  */

static int i=0;   /* Modul-global      */
```

```
int main(){

    int i=1, m;           /* lokal */
    printf("i=%d \n", i);
    printf("i=%d \n", calc_i());
    printf("i=%d \n", calc_i_ext());
    printf("k=%d \n", k);

    return 0;
}
```

```
int calc_i(){      /* Fkt.-Definition */
    return i;
}
```

Was wird ausgedruckt ?
Ist etwas falsch ?

```
/* -- Modul_2.c -- */


int k=15; /* global */
static int i=2;
```

```
int calc_i_ext(){
    return i;
}
```

```
/* -- Modul_2.h --*/

extern int k;
int calc_i_ext();
```