

U2 2. Übung

- Datentypen mit fester Größe
- Bitoperatoren
- Funktionen
- Nachtrag Compiler
- Aufgabe 2

U2-1 Datentypen fester Größe

U2-1 Datentypen fester Größe

- Die Größe der primitiven Datentypen in C ist architekturabhängig
- Beispiel für drei Architekturen (Angaben in Bits)

	8-bit AVR	Intel x86-32	Intel x86-64
char	8	8	8
short	16	16	16
int	16	32	32
long	32	32	64

- Probleme
 - ◆ Portabilität des Quellcodes eingeschränkt
 - ◆ Insbesondere in der hardwarenahen Programmierung braucht man oft Datentypen einer bekannten, festen Größe (I/O-Register)

U2-1 Datentypen fester Größe

- C99 definiert neue Datentypen mit definierter Größe
- Auszug der wichtigsten Typen:

<code>int8_t</code>	8-bit signed
<code>uint8_t</code>	8-bit unsigned
<code>int16_t</code>	16-bit signed
<code>uint16_t</code>	16-bit unsigned
<code>int32_t</code>	32-bit signed
<code>uint32_t</code>	32-bit unsigned

- verfügbar durch Einbinden von **stdint.h**
 - ☞ wird auch mit vielen nicht C99-konformen Compilern geliefert
 - ☞ kann ansonsten auch relativ einfach selbst erstellt werden
- die Größe von Zeigertypen ist immer architekturabhängig (Adressbusbreite)

U2-2 Bitweise Operatoren

- Logische Operatoren:

"nicht"

!	
f	w
w	f

"und"

&&	f	w
f	f	f
w	f	w

"oder"

 	f	w
f	f	w
w	w	w

U2-2 Bitweise logische operatoren

■ Binäre Operatoren

"nicht"	"und"	"oder"	"x-oder"
\sim	$\&$	$ $	\wedge
$\begin{array}{c c} & \\ \hline f & w \\ w & f \end{array}$	$\begin{array}{c cc} & f & w \\ \hline f & f & f \\ w & f & w \end{array}$	$\begin{array}{c cc} & f & w \\ \hline f & f & w \\ w & w & w \end{array}$	$\begin{array}{c cc} & f & w \\ \hline f & f & w \\ w & w & f \end{array}$

■ Beispiel

\sim	$\&$	$ $	\wedge
	1100	1100	1100
1001	1001	1001	1001
0110	1000	1101	0101

U2-2 Shiftoperatoren

➡ Bits werden im Wort verschoben

<<	Links-Shift
>>	Rechts-Shift

■ Beispiel:

x	<table><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	1	0	0	1	1	1	0	0
1	0	0	1	1	1	0	0		
x << 2	<table><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	1	0	0	0	0
0	1	1	1	0	0	0	0		

U2-3 Funktionen

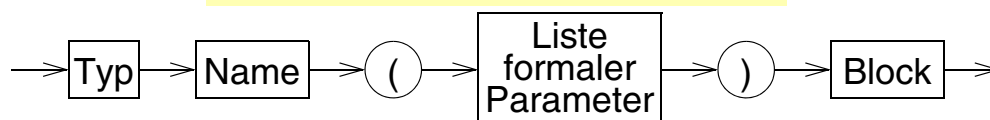
- ➔ Funktionen dienen der Abstraktion
- Name und Parameter abstrahieren
 - vom tatsächlichen Programmstück
 - von der Darstellung und Verwendung von Daten
- Verwendung
 - ◆ mehrmals benötigte Programmstücke können durch Angabe des Funktionsnamens aufgerufen werden
 - ◆ Schrittweise Abstraktion (**Top-Down**- und **Bottom-Up**-Entwurf)
 - ◆ Strukturierung

U2-3 Funktionsdefinition

- Schnittstelle (Typ, Name, Parameter) und die Implementierung

- ◆ Beispiel:

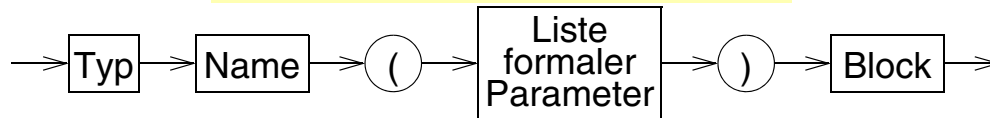
```
int addition ( int a, int b ) {
    int ergebnis;
    ergebnis = a + b;
    return ergebnis;
}
```



- Typ
 - ◆ Typ des Werts, der am Ende der Funktion als Wert zurückgegeben wird
 - ◆ beliebiger Typ
 - ◆ `void` = kein Rückgabewert
- Name
 - ◆ beliebiger Bezeichner, kein Schlüsselwort

U2-3 Funktionsdefinition (2)

```
int addition ( int a, int b ) {}
```



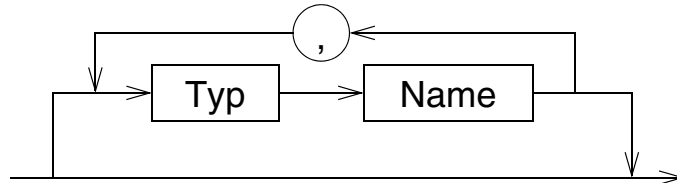
Liste formaler Parameter

◆ **Typ:** beliebiger Typ

◆ **Name:**
beliebiger Bezeichner

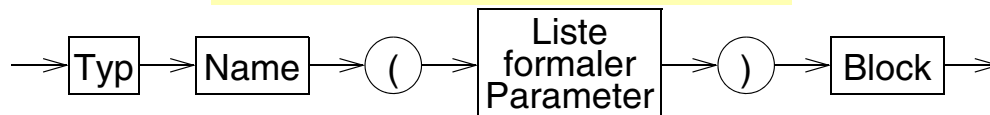
◆ die formalen Parameter stehen innerhalb der Funktion für die Werte, die beim Aufruf an die Funktion übergeben wurden (= **aktuelle Parameter**)

◆ die formalen Parameter verhalten sich wie Variablen, die im **Funktionsrumpf** definiert sind und mit den aktuellen Parametern vorbelegt werden



U2-3 Funktionsdefinition (3)

```
int addition ( int a, int b ) {}
```



Block

◆ beliebiger Block

◆ zusätzliche Anweisung

```
return ( Ausdruck );
```

oder

```
return;
```

bei void-Funktionen

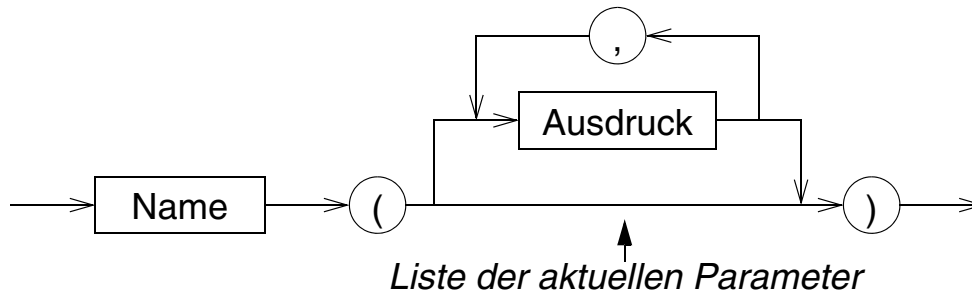
- Rückkehr aus der Funktion: das Programm wird nach dem Funktionsaufruf fortgesetzt
- der Typ des Ausdrucks muss mit dem Typ der Funktion übereinstimmen
- die Klammern können auch weggelassen werden

U2-3 Funktionsaufruf

- Aufruf einer Funktion aus dem Ablauf einer anderen Funktion

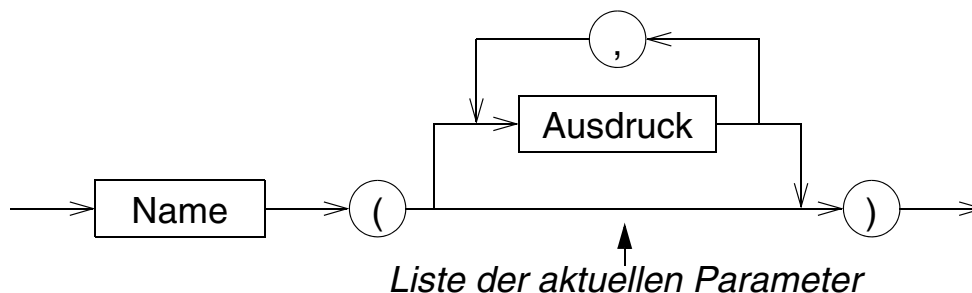
◆ Beispiel:

```
int main ( ) {
    int summe;
    summe = addition(3,4);
    ...
}
```



- Jeder Funktionsaufruf ist ein Ausdruck
- `void`-Funktionen können keine Teilausdrücke sein
 - ◆ wie Prozeduren in anderen Sprachen (z. B. Pascal)

U2-3 Funktionsaufruf (2)



- Die Ausdrücke in der Parameterliste werden ausgewertet, **bevor** in die Funktion gesprungen wird
 - ➔ **aktuelle Parameter**
- Anzahl und Typen der Ausdrücke in der Liste der aktuellen Parameter müssen mit denen der formalen Parameter in der Funktionsdefinition übereinstimmen
- Die Auswertungsreihenfolge der Parameterausdrücke ist **nicht** festgelegt

U2-3 Beispiel

```
float power (float b, int e)
{
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    return(prod);
}
```

```
float x, y;

y = power(2+x,4)+3;
```

≡

```
float x, y, power;
{
    float b = 2+x;
    int e = 4;
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    power = prod;
}
y=power+3;
```

U2-3 Regeln

- Funktionen werden global definiert
 - ➔ keine lokalen Funktionen/Prozeduren wie z. B. in Pascal
- `main()` ist eine normale Funktion, die aber automatisch als erste beim Programmstart aufgerufen wird
 - Ergebnis vom Typ `int` - wird an die Shell zurückgeliefert (in Kommandoprozeduren z. B. abfragbar)
- rekursive Funktionsaufrufe sind zulässig
 - ➔ eine Funktion darf sich selbst aufrufen (z. B. zur Fakultätsberechnung)

```
fakultaet(int n)
{
    if ( n == 1 )
        return(1);
    else
        return( n * fakultaet(n-1) );
}
```

U2-3 Regeln (2)

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
 - = Rückgabotyp und Parametertypen müssen dem Compiler bekannt sein
 - ◆ durch eine Funktionsdefinition ist die Funktion automatisch auch deklariert
- Funktionsdeklaration
 - ◆ soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine **Deklaration** bekannt gemacht werden
 - ◆ Syntax:

Typ Name (Liste formaler Parameter);

 - Parameternamen können weggelassen werden, die Parametertypen müssen aber angegeben werden!
 - ◆ Beispiel:


```
double sinus(double);
```

U2-3 Funktionsdeklarationen — Beispiel

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

double sinus(double);
/* oder: double sinus(double x); */

int main()
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
        wert, sinus(wert));
    return(0);
}
```

```
double sinus (double x)
{
    double summe;
    double x_quadrat;
    double rest;
    int k;

    k = 0;
    summe = 0.0;
    rest = x;
    x_quadrat = x*x;

    while ( fabs(rest) > 1e-9 ) {
        summe += rest;
        k += 2;
        rest *= -x_quadrat/(k*(k+1));
    }
    return(summe);
}
```


U2-3 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
 - call by value
 - call by reference

call by value

- Normalfall in C
- Es wird eine Kopie des aktuellen Parameters an die Funktion übergeben
 - ➔ die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
 - ➔ die Funktion kann den Wert des formalen Parameters (also die Kopie!) ändern, ohne dass dies Auswirkungen auf den Wert des aktuellen Parameters beim Aufrufer hat
 - ➔ die Funktion kann über einen Parameter dem Aufrufer keine Ergebnisse mitteilen

U2-3 Parameterübergabe an Funktionen (2)

call by reference

- In C nur indirekt mit Hilfe von Zeigern realisierbar
- Der Übergabeparameter ist eine Variable und die aufgerufene Funktion erhält die Speicheradresse dieser Variablen
 - ➔ die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
 - ➔ wenn die Funktion den Wert des formalen Parameters verändert, ändert sie den Inhalt der Speicherzelle des aktuellen Parameters
- ➔ auch der Wert der Variablen (aktueller Parameter) beim Aufrufer der Funktion ändert sich dadurch

U2-4 Nachtrag Compilerparameter

- Verwendung des Parameter: -Werror
 - ◆ Wandelt Warnungen in Fehler um
 - ◆ Der Compilervorgang wird auch bei Warnungen abgebrochen
 - ◆ Die meisten Warnungen sind echte Fehler.