

Systemnahe Programmierung

{Programmierung in C}

Prof. Dr. Stefan Böhmer

stefan.boehmer@its.h-brs.de



SS 2018



- 1. Einführung in die Programmiersprache C**
- 2. Gültigkeitsbereiche, komplexe Datentypen**
- 3. Kontrollstrukturen, Ein- und Ausgabe**
- 4. Zeiger, Felder und Zeichenketten**
- 5. Makros, C-Entwicklungswerkzeuge**
- 6. Dateisystem**
- 7. Ausgewählte Beispiele**
(Prozesse, Threads, ...)

Materialien zur Vorlesung

Verfügbar auf der E-Learning Plattform der Hochschule


➦ <https://lea.h-brs.de>

Benutzername: <FB 02 Kürzel>

Passwort: <Kennwort für die Bibliothek>

- Folien
 - Übungsaufgaben
 - Beispielquelltexte (keine Musterlösungen)
 - Hilfen (Compiler, IDE, weitere Übungsaufgaben...)
- ⇒ nächste Folie


Materialien zur Vorlesung -- Hilfen


 **Online-Kurs**
Inhalt geändert


[Inhalt](#) [Info](#) [Einstellungen](#) [Lernfortschritt](#) [Export](#) [Rechte](#)


[Zeigen](#) [Verwalten](#) [Sortierung](#) [Seite gestalten](#)

Inhalt

 **C - Compiler**
Installations- und Anwendungsanleitungen für C-Compiler unter Windows und Linux
Typ: Lernmodul HTML

 **C - Dokumentation**
Mini Dokumentation für die grundlegende C-Syntax und einige C-Funktionen im Vergleich zu Java
Typ: Lernmodul HTML

 **C - Entwicklungsumgebungen**
Installations- und Anwendungsanleitungen für C-Entwicklungsumgebungen unter Windows und Linux
Typ: Lernmodul HTML

 **C - Übungsaufgaben**
Empfohlene Übungsaufgaben zur Klausurvorbereitung
Typ: Lernmodul HTML

Literatur

- S.P. Harbison and G. L. Steele , *C: A Reference Manual*,
Prentice Hall; 5 edition (March 3, 2002),
ISBN-13: 978-0130895929
- J. Wolf , *C von A-Z*,
Galileo Press; 3. Auflage (2009), ISBN-13: 978-3-8362-1411-7
- ISO/IEC 9899, Programming Languages,
<http://www.open-std.org/jtc1/sc22/wg14>
- ...

Vorteile:

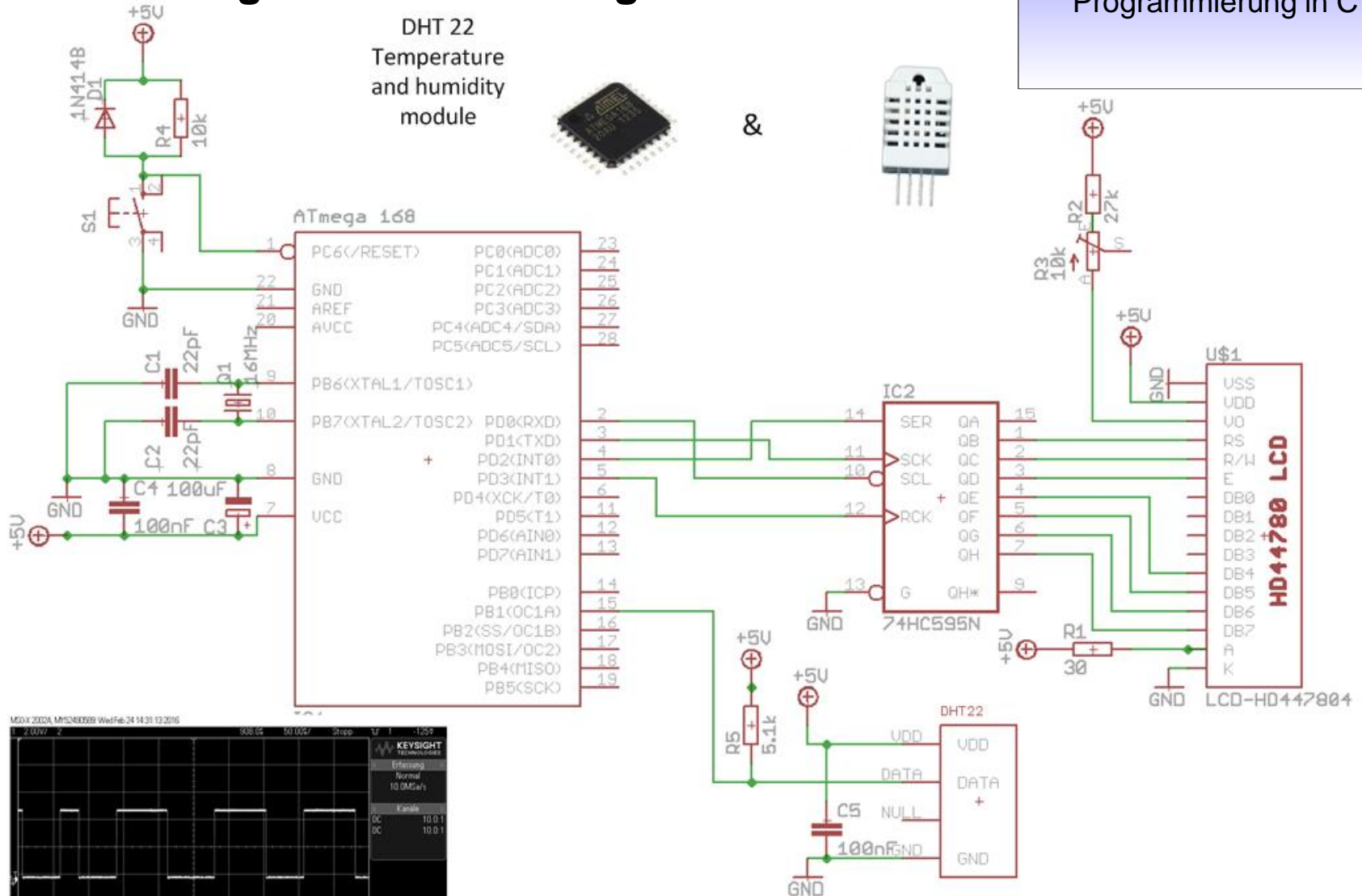
- Universell, für (fast) alle Systeme existieren Entwicklungsumgebungen
- Effiziente (hardwarenahe) Programmierung
- Portabilität
- Umfangreiche (Standard-) Bibliotheken
- Modularität

Nachteile:

- Syntaktische Schwachstellen (z.B. „==“ und „=“)
- Hohe Freiheitsgrade bei der Programmierung ↗ pot. Probleme: unlesbarer Code, implizite Typkonvertierung
- Objektorientierte Konzepte werden nicht unterstützt ⇒ C++

Anwendung: Internet of Things?!

Einführung –
Programmierung in C



- 1. Einführung in die Programmiersprache C**
2. Gültigkeitsbereiche, komplexe Datentypen
3. Kontrollstrukturen, Ein- und Ausgabe
4. Zeiger, Felder und Zeichenketten
5. Makros, C-Entwicklungswerkzeuge
6. Dateisystem
7. Ausgewählte Beispiele
(Prozesse, Threads, ...)

Einleitung

- Ein C-Programm besteht aus einer oder **mehreren** Textdatei(en) ohne Formatierung, die mit verschiedenen Editoren bearbeitet werden können
- Bei der Übersetzung in Maschinensprache (Binärcode) wird zwischen Groß- und Kleinschreibung unterschieden
- Trennzeichen (= White-Spaces) im Quelltext, wie z.B. Leerzeichen, Tabulator oder Zeilenumbruch werden ignoriert
- Üblicherweise werden größere C-Programme modular aufgebaut
Eine Quellcode-Datei wird dabei als **Modul** bezeichnet

Aufbau eines C-Programms

Gliederung und Funktionen

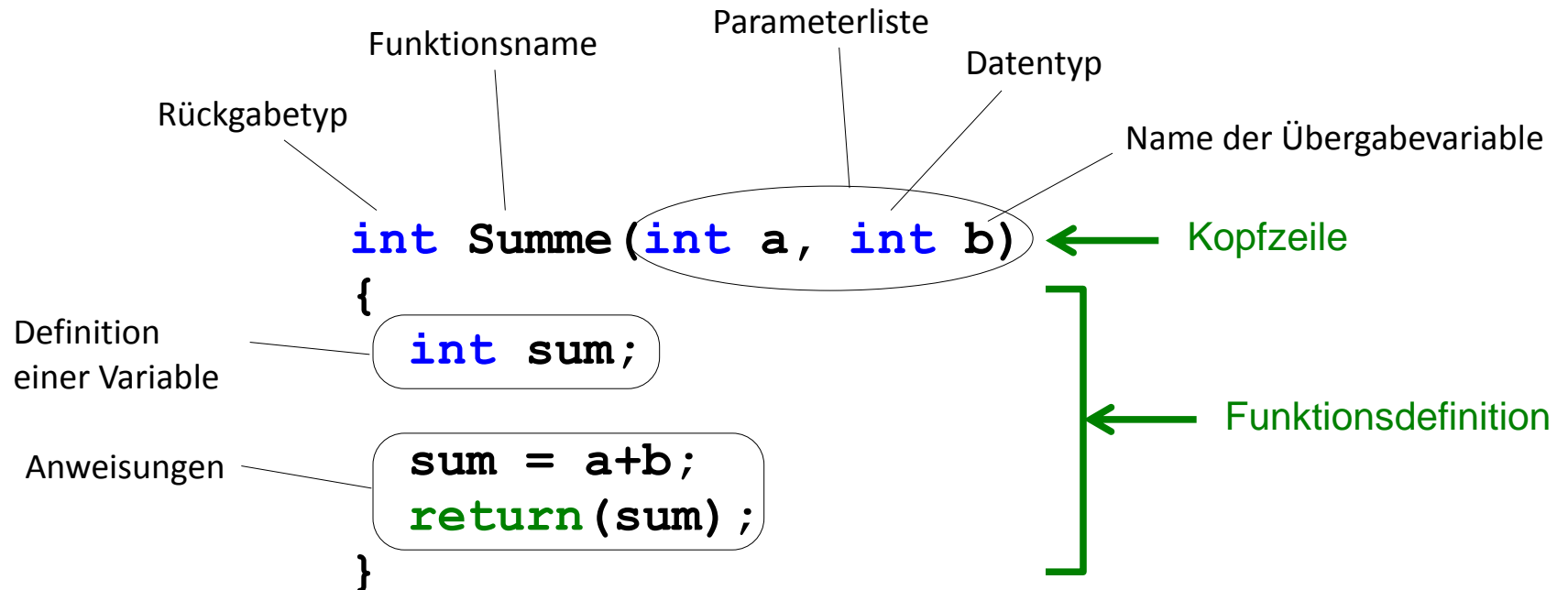
1. Einführung in die Programmiersprache C

- Ein C-Programm ist eine Sammlung von **Funktionen** (prozedurale Programmiersprache)
- Jede Funktion besteht aus einer Kopfzeile und einer Funktionsdefinition
- Kopfzeile besteht aus:
 - Datentyp des Rückgabewertes
 - Funktionsname
 - Liste der Übergabeparameter
- Funktionsdefinition:
 - wird durch '{' und '}' geklammert (Block)
 - enthält die Definition lokaler Variablen
 - enthält Anweisungen
- Es muss **genau eine** Funktion mit dem Namen **main()** existieren. Diese Funktion wird vom Betriebssystem beim Programmstart aufgerufen

Ein erstes Beispiel

Struktur einer Funktion

1. Einführung in die Programmiersprache C



Eine Bibliotheksfunktion

printf()

1. Einführung in die Programmiersprache C

- Die Funktion `printf` wird zur Ausgabe von Text und Variablenwerten auf dem Bildschirm verwendet
- Der erste Parameter der Parameterliste ist eine Spezifikation des Ausgabeformates und des auszugebenden Textes. Für Variablenwerte werden Platzhalter im Text vorgesehen, die zugehörigen Variablen werden in nachfolgenden Positionen der Parameterliste angegeben

- Beispiel:

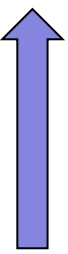
```
printf ("Hier steht der Ausgabertext Nr. %d", zaehler);
```



Ausgabeformat und
Ausgabertext



Platzhalter für die
Ausgabe eines
Variablenwertes



Variable deren Wert
ausgegeben werden
soll

Ein erstes Beispiel

Ein vollständiges C-Programm

```
int printf(const char *string,...);

int Summe(int a, int b)
{
    int sum;

    sum = a + b;
    return(sum);
}

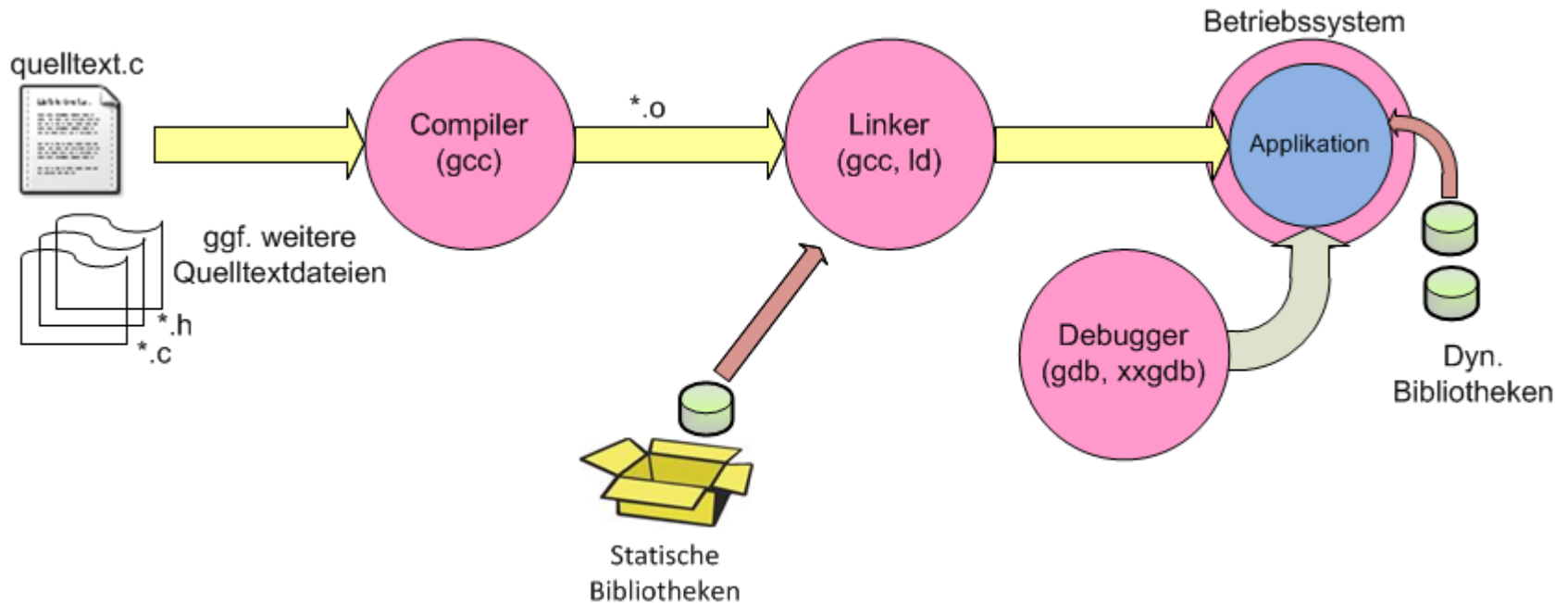
int main(int argc, char **argv)
{
    int a = 1,
        b = 3;
    int sum;

    sum = Summe(a,b);
    printf("Ergebnis: %d\n", sum);
    return(0);
}
```

Vorab-Information: Entwicklungsumgebung

...vom Quelltext zum ausführbaren Programm...

1. Einführung in die Programmiersprache C



Beispiel:

Erzeugen eines ausführbaren Programms

Übersetzen einer Quelltext-Datei „quelltext.c“

```
# gcc -c quelltext.c
```

Binden einer Binärdatei (quelltext.o) zu einem ausführbaren Programm
(Bsp.-Name: prog.exe)

```
# gcc quelltext.o -o prog.exe
```

Beide Arbeitsschritte durch einen Programmaufruf realisieren:

```
# gcc quelltext.c -o prog.exe
```

besser („-Wall“ = alle Warnungen auf dem Bildschirm ausgeben):

```
# gcc -Wall quelltext.c -o prog.exe
```


Strukturelemente eines C Programms

- **Bezeichner** werden als Namen für Funktionen und Variablen verwendet
- Bezeichner sind zusammenhängende Zeichenketten ohne Leerzeichen
- Erlaubt sind: Buchstaben, Ziffern und das Zeichen '_' (Underscore)
- Das erste Zeichen darf keine Ziffer sein
- Groß und Kleinbuchstaben werden unterschieden
- Die ersten 31 Zeichen sind signifikant
- Coding-Convention:
 - Makros: nur Großbuchstaben
 - Funktionen/Variablen: Groß- und Kleinbuchstaben gemischt
 - ggf. Modulbezeichnung voranstellen
 - ggf. Typ- oder Speicherklassen-Kennzeichnung voranstellen/anhängen
- Beispiele:

```
#define MAXIMUM 12           // Makro
int zaehler1;               // Integer Variable
int *modul_anzahl_ptr;      // Zeiger Variable
int modul_getZaehler(void); // Funktion
```

- Einige Bezeichner sind als Schlüsselwort reserviert und dürfen nicht verwendet werden:

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

- Es gibt einige vordefinierte Bezeichner (erkennbar am doppelten Unterstrich), die als Makro (vgl. Präprozessor) definiert sind:

<code>__LINE__</code>	Nummer der Zeile im Quellcode
<code>__FILE__</code>	Name des Quellcodes
<code>__DATE__</code>	Datum der Übersetzung
<code>__TIME__</code>	Uhrzeit der Übersetzung
<code>__func__</code>	Name der Funktion

- Elementare Datentypen:

char	Zeichen aus dem Zeichensatz der Maschine
int	vorzeichenbehafteter ganzzahliger Wert, üblicherweise in Maschinen-Wortbreite
float	Gleitpunktwert mit einfacher Genauigkeit
double	Gleitpunktwert mit doppelter Genauigkeit
void	Leerer bzw. unspezifischer Datentyp

- Modifikationen:

- in Verbindung mit **char** oder **int**:

unsigned	vorzeichen <u>lose</u> Ganzzahl
signed	vorzeichen <u>behaftete</u> Ganzzahl

- in Verbindung mit **int** oder **double**:

short	geringere Wortbreite (nur bei int erlaubt)
long	höhere Genauigkeit bzw. größere Wortbreite

- Datentypen und Modifikatoren müssen nicht immer vollständig angegeben werden.
Es gelten die Entsprechungen:

signed	= signed int	= int
unsigned	= unsigned int	
short	= short int	
long	= long int	

- Beispiele:
 unsigned char byte;
 short word;
 long double prec128;
- Tatsächliche Speicherallokation ist vom Compiler/Hardware abhängig!!!
Die Wortbreiten sind in der Definitionsdatei **<limits.h>** und **<float.h>** angegeben.
- Der Speicherbedarf eines Datentyps kann mit der Funktion **sizeof()** ermittelt werden

- Kommentare werden durch die Zeichenfolge `/*` begonnen und enden mit `*/`
- Einzeilige Kommentare werden mit `//` eingeleitet und enden mit dem Zeilenumbruch
- Verschachtelte Kommentare sind nicht erlaubt, werden aber von einigen C-Compilern unterstützt

- Beispiele:

```
/* Dies ist ein Kommentar */
```

```
// Einzeiliger Kommentar in C++ (C), endet mit Zeilenumbruch
```

```
/* Dieser Kommentar benötigt  
mehrere Zeilen */
```

```
/* Bei sehr langen Kommentaren kann man  
* noch ein paar '*' Zeichen einfügen, dann ist der  
* Quelltext besser lesbar  
*/
```

```
/* Verschachtelte /* Kommentare */ werden nicht von allen  
Compilern unterstützt */
```

- Alle Funktionen und Variablen müssen vor ihrer ersten Verwendung deklariert bzw. definiert werden
- **Deklaration**: Variable/Funktion bekannt geben
Definition: Variable/Funktion anlegen (es wird Speicher reserviert)
- Deklaration einer Funktion \equiv Funktions-Prototyp
- Beispiele:

```
extern int myFunctionA(int , int ); // Funktions-Prototyp
```

```
int myFunctionB(int a) // Definition einer Funktion
{
    return(a*a);
}
```

```
int a=1, b; // Definition
```

```
extern int x; // Deklaration
```

- Ausdrücke bestehen aus Konstanten, Variablen und/oder Funktionen, die durch Operatoren miteinander verknüpft sind, z.B.:

```
3 * 5 + 2
x * sqrt( y ) - sin(alpha)
(a == 3) && (b != 0)
```

- Die Reihenfolge, mit der die Ausdrücke ausgewertet/bearbeitet werden, ist festgelegt durch:
 - Priorität der Operatoren
 - Assoziativität (= Auswertungsreihenfolge) der Operatoren
- Durch Klammerung kann (und sollte bei geringstem Zweifel) die Reihenfolge eindeutig definiert werden
- Ausdrücke geben einen Wert zurück, dessen Typ von der Operation abhängt
- Anweisungen sind Ausdrücke, die durch ein ';' (Semikolon) abgeschlossen sind, z.B.:

```
x = sin(2.3) ;
printf("fertig!\n") ;
```


Operatoren (1/4)

Arithmetische Operatoren

1. Einführung in die Programmiersprache C

Operator	Bezeichnung	Priorität	Assoziativität	Typ
-	Vorzeichen	15	rechts	unär
++	Inkrement	15/16*	rechts	unär
--	Dekrement	15/16*	rechts	unär
*	Multiplikation	13	links	binär
/	Division	13	links	binär
%	Modulo	13	links	binär
+	Addition	12	links	binär
-	Subtraktion	12	links	binär
=	Zuweisung	2	rechts	binär
op=	zusammengesetzte Zuweisung	2	rechts	binär
()	Klammer	16	links	unär

Beispiele:

Anwendung und Priorität/Assoziativität

`int a=1, b=2, c=3, d=0, e=4;`

`d += a;` entspricht (ohne Prio./Ass) `d = d + a;` \Rightarrow Wert von d jetzt: 1

`e++;` entspricht (ohne Prio./Ass) `e = e + 1;` \Rightarrow Wert von e jetzt: 5

`a = b += c :`

1. Operator `=` und `+=` haben dieselbe Priorität, sind also gleichberechtigt
 2. Assoziativität: `=` ist rechts-assoziativ \Rightarrow zunächst wird der Ausdruck `b+=c` ausgewertet
 3. Der Operator `+=` ist ebenfalls rechts-assoziativ \Rightarrow zunächst wird der Wert von c bestimmt
- \Rightarrow Wert von b ist jetzt: 5
- \Rightarrow Wert von a ist jetzt: 5

Operatoren (2/4)

Vergleichs- und Boolesche-Operatoren

1. Einführung in die Programmiersprache C

Operator	Bezeichnung	Priorität	Assoziativität	Typ
<	kleiner	10	links	binär
<=	kleiner gleich	10	links	binär
>	größer	10	links	binär
>=	größer gleich	10	links	binär
==	gleich	9	links	binär
!=	ungleich	9	links	binär
!	NICHT (Boolsche Algebra)	15	rechts	unär
&&	UND (Boolsche Algebra)	5	links	binär
	ODER (Boolsche Algebra)	4	links	binär

Operatoren (3/4)

Operatoren zur Bitmanipulation

1. Einführung in die Programmiersprache C

Operator	Bezeichnung	Priorität	Assoziativität	Typ
~	NICHT (bitweise)	15	rechts	unär
&	UND (bitweise)	8	rechts	binär
^	Exklusiv Oder (bitweise)	7	links	binär
	ODER (bitweise)	6	links	binär
<<	Linksschieben	11	links	binär
>>	Rechtsschieben	11	links	binär

Operatoren (4/4)

Sonstige Operatoren

1. Einführung in die Programmiersprache C

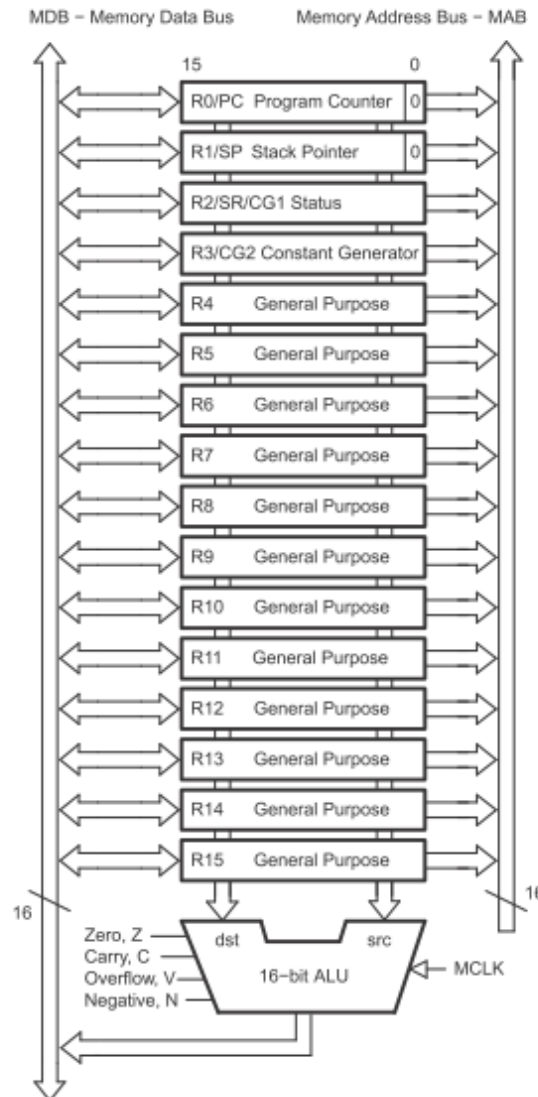
Operator	Bezeichnung	Priorität	Assoziativität	Typ
[]	Feldindex	16	links	unär
->	Zeiger auf Struktur	16	links	unär
.	Strukturelement	16	links	binär
&	Adresse	15	rechts	unär
,	Trennung	1	links	binär
? :	Auswahl	3	rechts	tertiär
(Datentyp)	Typumwandlung	14	rechts	unär
*	Zeiger De-Referenzierung	15	rechts	unär
sizeof()	Datentyp-Größe bestimmen	15	links	unär

1. Einführung in die Programmiersprache C ☒
- 2. Gültigkeitsbereiche, komplexe Datentypen**
3. Kontrollstrukturen, Ein- und Ausgabe
4. Zeiger, Felder und Zeichenketten
5. Makros, C-Entwicklungswerkzeuge
6. Dateisystem
7. Ausgewählte Beispiele
(Prozesse, Threads, ...)

Aufbau CPU/Zentraleinheit

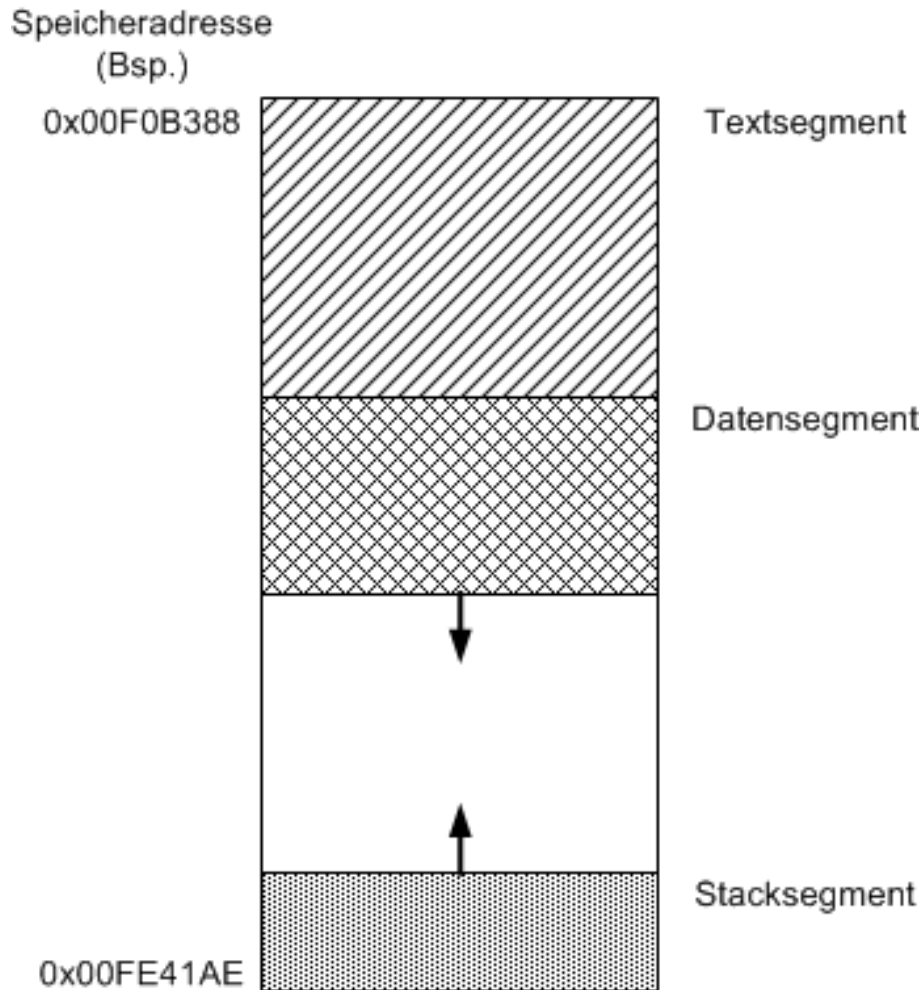
Bsp.: Texas Instruments MSP 430

2. Gültigkeitsbereiche,
komplexe Datentypen



Speicherbelegung durch einen Prozess

2. Gültigkeitsbereiche, komplexe Datentypen



- Betriebssystem (Lader) lädt ein relocierbares Programm in den Hauptspeicher (RAM)
- Der ausführbare Programmcode ist im Textsegment (Code Segment) abgelegt
- Daten- und Stacksegment beinhalten die Daten der (temporären) Variablen
- Ggf. wird (für bestimmte dyn. Variablen) auch ein Heap-Segment verwendet

Definition von Variablen

Geltungsbereiche und Speicherklassen

- Variablen können unterschiedliche **Geltungsbereiche** und unterschiedliche **Speicherklassen** haben
- Geltungsbereiche:

lokal	-	nur in der Funktion bekannt
modulglobal	-	nur im Modul bekannt
programmglobale	-	im gesamten Programm bekannt
- Speicherklassen:

Die Zuordnung (Ablage) von Variablen und Konstanten zu Segmenten erfolgt durch den Compiler/Linker:

 - lokale Variablen \Rightarrow Stacksegment (SS)
 - modulglobale (programmglobale) und statisch lokale Variablen
 \Rightarrow Datensegment (DS)
 - Konstanten dürfen nicht geändert werden
 \Rightarrow Textsegment (CS), Datensegment (DS)

Durch **Attribute** kann die Segmentzuordnung gesteuert werden

Definition von Variablen

Attribute

2. Gültigkeitsbereiche,
komplexe Datentypen

- auto** Compiler legt Speicherklasse fest (default)
- register** Empfehlung an Compiler: Objekt soll in einem Register abgelegt werden
{Anm.: Zugriff über Adress-Operator (Zeiger) nicht möglich}
- static** Innerhalb einer Funktion: Variable liegt an festem Speicherplatz (DS),
sie behält ihren Wert bis zum Wiedereintritt
Außerhalb jeder Funktion: Variable ist modulglobal, ihr Name
kollidiert nicht mit gleichnamigen Variablen aus anderen Modulen
- extern** Deklariert eine Variable als extern. Die Variable muss in einem
anderen Modul angelegt sein (Referenzdeklaration)
- volatile** Verhindert Compileroptimierungen, da die Variable auch durch
andere Prozesse (Threads) verändert werden kann
- const** Variable kann nicht verändert werden

Definition von Variablen

Beispiele für Geltungsbereiche

2. Gültigkeitsbereiche,
komplexe Datentypen

```
/* calc.c */

/* global */
unsigned char fehler = 0;

/* modulglobal */
static unsigned char isNull(float a)
{
    if(a == 0) {
        return(1);
    }
    else {
        return(0);
    }
}

/* global */
float div(float a, float b)
{
    /* lokal, fluechtig */
    float ergebnis = 0;

    if(isNull(b)) {
        fehler = 1;
    }
    else {
        ergebnis = a/b;
    }
    return(ergebnis);
}
```

```
/* mathe.c */

/* Deklaration einer externen Variable */
extern unsigned char fehler;

/* Deklaration einer externen Funktion */
extern float div(float a, float b);

/* modulglobal */
static void Fehlermeldung(void);
static float ergebnis;

void main()
{
    ergebnis = div(2.0, 3.0);
    if(fehler == 1) {
        Fehlermeldung();
    }
}

static void Fehlermeldung(void)
{
    /* lokal, statisch */
    static unsigned zaehler = 0;

    zaehler = zaehler + 1;

    printf("Bisher: %u Fehler\n", zaehler);
}
```

- Es dürfen Operanden mit unterschiedlichen Datentypen verknüpft werden
- Implizite Typumwandlung
 - Der Compiler nimmt die Typumwandlung vor
 - Der „kleinere“ Datentyp (z.B. `int`) wird in den „größeren“ Datentyp (z.B. `long`) umgewandelt. Das Ergebnis ist ebenfalls vom „größeren“ Datentyp
Ausnahme: Bei einer Zuweisung wird in den Datentyp des linken Operanden umgewandelt
- Explizite Typumwandlung:
Konvertierung mit dem cast-Operator
Syntax:
(`<Datentyp>`) Ausdruck
- Coding-Convention:
⇒ Implizite Typumwandlung vermeiden

Hinweise zu Operatoren und Datentypen

Beispiele zur Typumwandlung

2. Gültigkeitsbereiche,
komplexe Datentypen

```
short s; // 2 Byte
int    i; // 4 Byte
float f; // 4 Byte

int    d = 1024;

i = 2*3.1;      i = 6
f = 2*3.1;      f = 6.2
s = d*d;        s = 0 (!!!)
```

```
/* cast.c */
```

```
signed    int wert  = -3;
unsigned int limit = 10;

if( wert < limit )
{
    printf("wert zu klein\n");
}
else
{
    printf("wert ok\n");
}
```

besser:

```
/* cast.c */
```

```
signed    int wert  = -3;
unsigned int limit = 10;

if( wert < (signed)limit )
{
    printf("wert zu klein\n");
}
else
{
    printf("wert ok\n");
}
```

- Konstanten können sein:
 - Zahl (Ganzzahl, Gleitpunktwert) = numerische Konstante
 - Zeichen = Zeichenkonstante
 - Zeichenkette = String-Konstante (*wird später behandelt*)
- Ganzzahlige numerische Konstanten:
 - dezimal (Basis 10): beginnt mit einer von **0** verschiedenen Ziffer, z.B. **123**
 - oktal (Basis 8): beginnt mit einer **0** (**Fehlerquelle!!!**), z.B. **0123**
 - hexadezimal (Basis 16): beginnt mit **0x**, z.B. **0x1F23**
 - automatische Typzuweisung, explizite Festlegung durch Anhängen der Buchstaben **L** oder **U** möglich, z.B.
 - 123** ist vom Typ **int**
 - 123L** ist vom Typ **long**
 - 123UL** ist vom Typ **unsigned long**

- Gleitpunktkonstanten:
 - Dezimalzahl, der gebrochene Anteil wird durch ein Punkt getrennt, z.B.: **3.1415**
 - In der Exponentialdarstellung wird der Exponent durch ein **E** gekennzeichnet, z.B.: **9.78E-2** ($= 9,78 \cdot 10^{-2}$)
 - Gleitpunktkonstanten sind vom Typ **double**, Konvertierung möglich durch Anhängen von **F** (-> **float**) oder **L** (-> **long double**)

Komplexe Datentypen

Zusammengesetzte Datentypen

struct

2. Gültigkeitsbereiche,
komplexe Datentypen

- Neben den elementaren Datentypen können individuelle Datentypen (= Strukturen) definiert werden

- Syntax:

// Deklaration der Struktur

```
struct StrukturName
{
    <Datentyp>    Name ;
    ...
};
```

// Definition einer Variable

```
struct StrukturName strucVar;
```

- Strukturen können selbst wieder Elemente von Strukturen sein
- Der Zugriff auf ein Strukturelement erfolgt mit dem ' . ' -Operator (Punkt-Operator):

`variable.element`

Zusammengesetzte Datentypen

Beispiel struct

2. Gültigkeitsbereiche,
komplexe Datentypen

```
struct Motor_typ
{
    int leistung;
    int hubraum;
};

struct Fahrgestell_typ
{
    int  anzAchsen;
    int  gewicht;
};

struct Fahrzeug_typ
{
    int      gesamtgewicht;
    struct Motor_typ      motor;
    struct Fahrgestell_typ fahrgestell;
};
```

```
struct Fahrzeug_typ vwPolo;
```

```
vwPolo.gesamtgewicht      = 1750;
vwPolo.motor.leistung     = 35;
vwPolo.motor.hubraum      = 1890;
vwPolo.fahrgestell.anzAchsen = 2;
vwPolo.fahrgestell.gewicht = 1200;
```

Zusammengesetzte Datentypen

union

- Eine Union (Variante) ist eine Struktur mit alternativen Datenelementen
- Es wird soviel Speicher belegt, wie das größte Datenelement benötigt; der Speicherbereich für alle Varianten **beginnt bei derselben Adresse**
- Auf den Inhalt des Speicherbereiches kann mit den Varianten zugegriffen werden
- Syntax:

// Deklaration der Union

```
union UnionName
{
    <Datentyp> Name1; //Variante 1
    <Datentyp> Name2; //Variante 2
    ...
};
```

// Definition einer Variable

```
union UnionName uniVar;
```

- Der Zugriff auf die Elemente erfolgt wie bei einer Struktur

Zusammengesetzte Datentypen

Beispiel union

2. Gültigkeitsbereiche,
komplexe Datentypen

```
/* byteorder.c */

union MyUnion
{
    char OneByte;
    int FourBytes;
};

int main()
{

    union MyUnion UniVar;

    UniVar.FourBytes = 0x1A2B3C4D;
    printf("Value of UniVar.OneByte is %d\n",UniVar.OneByte );

    return 0;
}
```

Ausgabe: Value of UniVar.OneByte is 77

Zusammengesetzte Datentypen

enum

- Ein Aufzählungstyp **enum** ist ein Datentyp mit beschränktem Wertebereich in Form einer (geordneten) Liste, die durch Aufzählung festgelegt wird

- Syntax:

// Deklaration der Aufzählung

```
enum EnumName  
{  
    Name1,  
    Name2,  
    ...  
    NameN  
};
```

// Definition einer Variable

```
enum EnumName enuVar;
```

- Durch explizite Wertzuweisung können den Konstanten bestimmte Werte zugewiesen werden, z.B.: **Name5 = 1**,
- Die Konstanten werden als Integer-Konstanten behandelt

Zusammengesetzte Datentypen

Beispiel enum

```
enum Wochentage
{
    Mo=1, Di=-5, Mi,Do,Fr,Sa
};

int main()
{

    enum Wochentage tag;

    printf("Value of UniVar.OneByte is %d\n",UniVar.OneByte );
    tag = Mo;
    printf("Heute ist %d\n",tag);
    tag=Di;
    printf("Morgen ist %d\n",tag);
    tag=Mi;
    printf("Übermorgen ist %d\n",tag);
    return 0;
}
```

Ausgabe:

Heute ist 1

Morgen ist -5

Übermorgen ist -4

Zusammengesetzte Datentypen

Typisierung

- Durch Verwendung des Schlüsselwortes **typedef** kann eine Struktur, Union oder Aufzählung als Datentyp registriert werden
- Syntax:

// Registrierung des Datentyps

```
typedef union
{
    <Datentyp> Name1; //Variante 1
    <Datentyp> Name2; //Variante 2
    ...
} UnionName;
```

// Definition einer Variable

```
UnionName uniVar;
```