D Einführung in die Programmiersprache C

D.1 C vs. Java

- Java: objektorientierte Sprache
  - ➤ zentrale Frage: aus welchen Dingen besteht das Problem
  - ➤ Gliederung der Problemlösung in Klassen und Objekte
  - ➤ Hierarchiebildung: Vererbung auf Klassen, Teil-Ganze-Beziehungen
  - ➤ Ablauf: Interaktion zwischen Objekten
- C: imperative / prozedurale Sprache
  - ➤ zentrale Frage: welche Aktivitäten sind zur Lösung des Problems auszuführen
  - ➤ Gliederung der Problemlösung in Funktionen
  - ➤ Hierarchiebildung: Untergliederung einer Funktion in Teilfunktionen
  - ➤ Ablauf: Ausführung von Funktionen

D.1 C vs. Java

1 C hat nicht

- Klassen und Vererbung
- Objekte
- umfangreiche Klassenbibliotheken

2 C hat

- Zeiger und Zeigerarithmetik
- Präprozessor
- Funktionsbibliotheken

D.2 Sprachüberblick

1 Erstes Beispiel

- Die Datei **hello.c** enthält die folgenden Zeilen:

```
/* say "hello, world" */
main()
{
    printf("hello, world\n");
}
```

- Die Datei wird mit dem Kommando **cc** übersetzt:

```
% cc hello.c        (C-Compiler)
oder
% gcc hello.c       (GNU-C-Compiler)
```

dadurch entsteht eine Datei **a.out**, die das ausführbare Programm enthält.

➤ ausführbares Programm liegt in Form von Maschinencode des Zielprozessors vor (kein Byte- oder Zwischencode)!

1 Erstes Beispiel (2)

- Mit der Option **-o** kann der Name der Ausgabedatei auch geändert werden – z. B.

```
% cc -o hello hello.c
```

- Das Programm wird durch Aufruf der Ausgabedatei ausgeführt:

```
% ./hello
hello, world
%
```

- Kommandos werden so in einem Fenster mit UNIX/Linux-Kommandointerpreter (Shell) eingegeben

➤ es gibt auch integrierte Entwicklungsumgebungen (z. B. Eclipse)

2 Aufbau eines C-Programms

- frei formulierbar **- *Zwischenräume*** *(Leerstellen, Tabulatoren, Newline und Kommentare*) werden i. a. ignoriert - sind aber zur eindeutigen Trennung direkt benachbarter Worte erforderlich
- ***Kommentar*** wird durch **`/*`** und **`*/`** geklammert
  keine Schachtelung möglich
- ***Identifier*** (Variablennamen, Marken, Funktionsnamen, ...) sind aus Buchstaben, gefolgt von Ziffern oder Buchstaben aufgebaut
  - "_" gilt hierbei auch als Buchstabe
  - Schlüsselwörter wie **`if`**, **`else`**, **`while`**, usw. können nicht als *Identifier* verwendet werden
  - ***Identifier*** müssen vor ihrer ersten Verwendung ***deklariert*** werden
- Anweisungen werden generell durch **`;`** abgeschlossen

3 Allgemeine Form eines C-Programms:

```
/* globale Variablen */
    ...

/* Hauptprogramm */
main(...)
{
    /* lokale Variablen */
        ...
    /* Anweisungen */
        ...
}

/* Unterprogramm 1 */
function1(...)
{
    /* lokale Variablen */
        ...
    /* Anweisungen */
        ...
}

/* Unterprogramm n */
functionN(...)
{
    /* lokale Variablen */
        ...
    /* Anweisungen */
        ...
}
```

4 wie ein C-Programm nicht aussehen sollte:

```
#define o define
#o ___o write
#o ooo (unsigned)
#o o_o_ 1
#o _o_ char
#o _oo goto
#o _oo_ read
#o o_o for
#o o_ main
#o o__ if
#o oo_ 0
#o _o(_,__,___)(void)___o(_,__,ooo(___))
#o __o(o_o_<<((o_o_<<(o_o_<<o_o_))+(o_o_<<o_o_)))
+(o_o_<<(o_o_<<(o_o_<<o_o_)))
o_(){_o_ _=oo_,__,___,____[__o];_oo ______;_____:___=__o-o_
_______:
_o(o_o_,____,__=(_-o_o_<___?_-
o_o_:___));o_o(;__;_o(o_o_,"\b",o_o_),__--);
_o(o_o_," ",o_o_);o__(--___)_oo
_______;_o(o_o_,"\n",o_o_);______:o__(_=_oo_(
oo_,____,__o))_oo _____;}
```

sieht eher wie Morse-Code aus, ist aber ein **gültiges** C-Programm.

D.3 Datentypen

- Datentypen
  - ➤ Konstanten
  - ➤ Variablen

→

- ◆ Ganze Zahlen
- ◆ Fließkommazahlen
- ◆ Zeichen
- ◆ Zeichenketten

1 Was ist ein Datentyp?

- Menge von Werten
  +
  Menge von Operationen auf den Werten
  - **Konstanten** Darstellung für einen konkreten Wert (`2`, `3.14`, `'a'`)
  - **Variablen** Namen für Speicherplätze, die einen Wert aufnehmen können

  ➥ Konstanten und Variablen besitzen einen Typ
- Datentypen legen fest:
  - Repräsentation der Werte im Rechner
  - Größe des Speicherplatzes für Variablen
  - erlaubte Operationen
- Festlegung des Datentyps
  - implizit durch Verwendung und Schreibweise (Zahlen, Zeichen)
  - explizit durch **Deklaration** (Variablen)

2 Standardtypen in C

- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert

`char`	Zeichen (im ASCII-Code dargestellt, 8 Bit)
`int`	ganze Zahl (16 oder 32 Bit)
`float`	Gleitkommazahl (32 Bit) etwa auf 6 Stellen genau
`double`	doppelt genaue Gleitkommazahl (64 Bit) etwa auf 12 Stellen genau
`void`	ohne Wert

2 Standardtypen in C (2)

- Die Bedeutung der Basistypen kann durch vorangestellte ***Typ-Modifier*** verändert werden

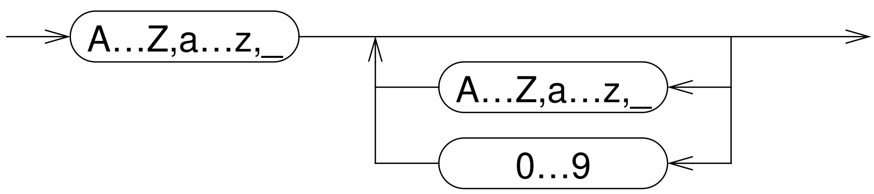
`short`, `long`
legt für den Datentyp **`int`** die Darstellungsbreite (i. a. 16 oder 32 Bit) fest.
Das Schlüsselwort **`int`** kann auch weggelassen werden

`long double`
`double`-Wert mit erweiterter Genauigkeit (je nach Implementierung) – mindestens so genau wie **`double`**

`signed`, `unsigned`
legt für die Datentypen **`char`**, **`short`**, **`long`** und **`int`** fest, ob das erste Bit als Vorzeichenbit interpretiert wird oder nicht

3 Variablen

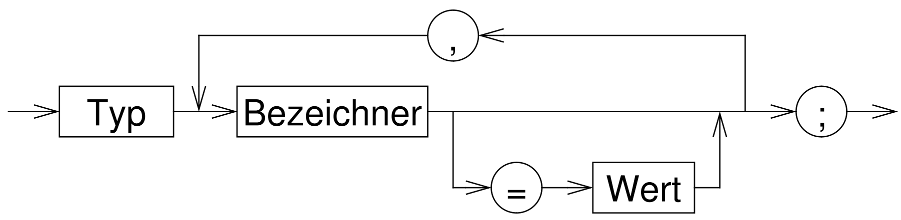
- Variablen haben:
  - **Namen** (Bezeichner)
  - Typ
  - zugeordneten Speicherbereich für einen Wert des Typs
    Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
  - **Lebensdauer**
    wann wird der Speicherplatz angelegt und wann freigegeben
- Bezeichner



(Buchstabe oder _ ,
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder _)

3 Variablen (2)

- Typ und Bezeichner werden durch eine **Variablen-Deklaration** festgelegt (= dem Compiler bekannt gemacht)
  - ◆ reine Deklarationen werden erst in einem späteren Kapitel benötigt
  - ◆ vorerst beschränken wir uns auf Deklarationen in **Variablen-Definitionen**
- eine **Variablen-Definition** deklariert eine Variable und reserviert den benötigten Speicherbereich



3 Variablen (3)

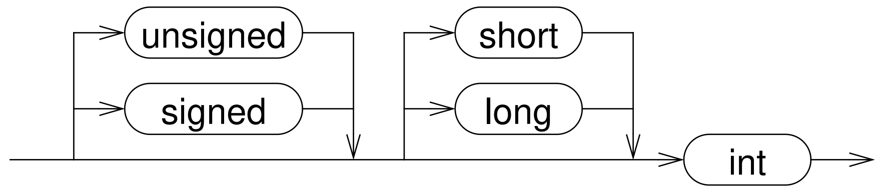
- Variablen-Definition: Beispiele

```
int a1;
float a, b, c, dis;
int anzahl_zeilen=5;
char Trennzeichen;
```

  - ◆ Position im Programm:
    - ➤ nach jeder "{"
    - ➤ außerhalb von Funktionen
    - ➤ neuere C-Standards und der GNU-C-Compiler erlauben Definitionen an beliebiger Stelle im Programmcode: Variable ab der Stelle gültig
- Wert kann bei der Definition initialisiert werden
- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar
- Lebensdauer ergibt sich aus der Programmstruktur

4 Ganze Zahlen

- Definition

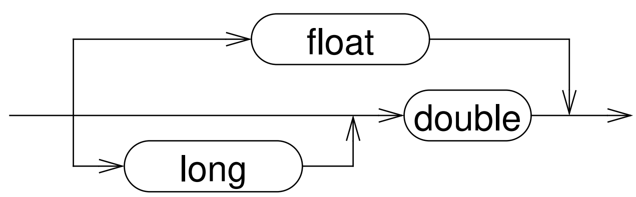


- Speicherbedarf(**short int**) ≤ Speicherbedarf(**int**) ≤ Speicherbedarf(**long int**)
- Speicherbedarf(**int**): meist 32 Bit
- Konstanten (Beispiele):

  42, -117
  035 (oktal = 29_{10})
  0x10 (hexadezimal = 16_{10})
  0x1d (hexadezimal = 29_{10})

5 Fließkommazahlen

- Definition



- Speicherbedarf(**float**) ≤ Speicherbedarf(**double**) ≤ Speicherbedarf(**long double**)
- Speicherbedarf(**float**): 32 Bit
- Konstanten (Beispiele):
  - ◆ normale Dezimalpunkt-Schreibweise

    3.14, **-2.718**, **368.345, 0.003**
    1.0 aber nicht einfach **1** (wäre eine **int**-Konstante!)
  - ◆ 10er-Potenz Schreibweise ($368.345 = 3.68345 \cdot 10^2$, $0.003 = 3.0 \cdot 10^{-3}$)

    3.68345e2, **3.0e-3**

6 Zeichen

- Bezeichnung: **char**
- Speicherbedarf: 1 Byte
- Repräsentation: ASCII-Code
  zählt damit zu den ganzen Zahlen
- Konstanten: Zeichen durch ' ' geklammert
  - Beispiele: **'a'**, **'X'**
  - Sonderzeichen werden durch **Escape-Sequenzen** beschrieben

Tabulator:	'\t'	Backslash:	'\\'
Zeilentrenner:	'\n'	Backspace:	'\b'
Apostroph:	'\''		

6 Zeichen (2)

American Standard Code for Information Interchange (ASCII)

NUL 00	SOH 01	STX 02	ETX 03	EOT 04	ENQ 05	ACK 06	BEL 07
BS 08	HT 09	NL 0A	VT 0B	NP 0C	CR 0D	SO 0E	SI 0F
DLE 10	DC1 11	DC2 12	DC3 13	DC4 14	NAK 15	SYN 16	ETB 17
CAN 18	EM 19	SUB 1A	ESC 1B	FS 1C	GS 1D	RS 1E	US 1F
SP 20	! 21	" 22	# 23	$ 24	% 25	& 26	' 27
(28	) 29	* 2A	+ 2B	, 2C	- 2D	. 2E	/ 2F
0 30	1 31	2 32	3 33	4 34	5 35	6 36	7 37
8 38	9 39	: 3A	; 3B	< 3C	= 3D	> 3E	? 3F
@ 40	A 41	B 42	C 43	D 44	E 45	F 46	G 47
H 48	I 49	J 4A	K 3B	L 3C	M 3D	N 3E	O 3F
P 50	Q 51	R 52	S 53	T 54	U 55	V 56	W 57
X 58	Y 59	Z 5A	[5B	\ 5C	] 5D	^ 5E	_ 5F
` 60	a 61	b 62	c 63	d 64	e 65	f 66	g 67
h 68	i 69	j 6A	k 6B	l 6C	m 6D	n 6E	o 6F
p 70	q 71	r 72	s 73	t 74	u 75	v 76	w 77
x 78	y 79	z 7A	{ 7B	| 7C	} 7D	~ 7E	DEL 7F

7 Zeichenketten (Strings)

- Bezeichnung: `char *`
- Speicherbedarf: (Länge + 1) Bytes
- Repräsentation: Folge von Einzelzeichen, letztes Zeichen: 0-Byte (ASCII-Wert 0)
- Werte: alle endlichen Folgen von `char`-Werten
- Konstanten: Zeichenkette durch `" "` geklammert
  - Beispiel: `"Dies ist eine Zeichenkette"`
  - Sonderzeichen wie bei char, `"` wird durch `\"` dargestellt
- Beispiel für eine Definition einer Zeichenkette:

```
char *Mitteilung = "Dies ist eine Mitteilung\n";
```


D.4 Ausdrücke

- Ausdruck = gültige Kombination von **Operatoren**, **Konstanten** und **Variablen**
- Reihenfolge der Auswertung
  - Die Vorrangregeln für Operatoren legen die Reihenfolge fest, in der Ausdrücke abgearbeitet werden
  - Geben die Vorrangregeln keine eindeutige Aussage, ist die Reihenfolge undefiniert
  - Mit Klammern `( )` können die Vorrangregeln überstimmt werden
  - Es bleibt dem Compiler freigestellt, Teilausdrücke in möglichst effizienter Folge auszuwerten

D.5 Operatoren

1 Zuweisungsoperator =

- Zuweisung eines Werts an eine Variable

- Beispiel:

```
int a;
a = 20;
```


2 Arithmetische Operatoren

- für alle **int** und **float** Werte erlaubt

+	Addition
–	Subtraktion
*	Mulitiplikation
/	Division
%	Rest bei Division, (modulo)
unäres –	negatives Vorzeichen (z. B. **–3**)
unäres +	positives Vorzeichen (z. B. **+3**)

- Beispiel:

```
a = -5 + 7 * 20 - 8;
```

3 spezielle Zuweisungsoperatoren

- Verkürzte Schreibweise für Operationen auf einer Variablen

  a *op*= b ≡ a = a *op* b

  mit ***op*** ∈ { +, –, *, /, %, <<, >>, &, ^, | }

- Beispiele:

```
a = -8;
a += 24;            /* -> a: 16 */
a /= 2;             /* -> a: 8 */
```


4 Vergleichsoperatoren

`<`	kleiner
`<=`	kleiner gleich
`>`	größer
`>=`	größer gleich
`==`	gleich
`!=`	ungleich

- **Beachte!** Ergebnistyp **`int`**: wahr (true) = 1
  falsch (false) = 0

- Beispiele:

```
a > 3
a <= 5
a == 0
if ( a >= 3 ) { …
```

5 Logische Operatoren

➥ Verknüpfung von Wahrheitswerten (wahr / falsch)

"nicht"

!	
f	w
w	f

"und"

&&	f	w
f	f	f
w	f	w

"oder"

||	f	w
f	f	w
w	w	w

◆ Wahrheitswerte (Boole'sche Werte) werden in C generell durch int-Werte dargestellt:

- ➤ Operanden in einem Ausdruck: Operand = 0: falsch; Operand ≠ 0: wahr
- ➤ Ergebnis eines Ausdrucks: falsch: 0; wahr: 1

5 Logische Operatoren (2)

■ Beispiel:

```
a = 5; b = 3; c = 7;
a > b && a > c
```

$$\underbrace{\underbrace{a > b}_{1} \;\textit{und}\; \underbrace{a > c}_{0}}_{0}$$

■ Die Bewertung solcher Ausdrücke wird abgebrochen, sobald das Ergebnis feststeht!

$$\underbrace{(a > c)}_{0} \;\&\&\; \underbrace{((d=a) > b)}_{\text{wird nicht ausgewertet}}$$

Gesamtergebnis=*falsch*

(d=a) wird nicht ausgeführt

6 Bitweise logische Operatoren

➥ Operation auf jedem Bit einzeln (Bit 1 = wahr, Bit 0 = falsch)

"nicht"	**~**
"und"	**&**
"oder"	**|**

Antivalenz "exklusives oder"

^	f	w
f	f	w
w	w	f

■ Beispiele:

x	1	0	0	1	1	1	0	0
~x	0	1	1	0	0	0	1	1
7	0	0	0	0	0	1	1	1
x | 7	1	0	0	1	1	1	1	1
x & 7	0	0	0	0	0	1	0	0
x ^ 7	1	0	0	1	1	0	1	1

7 Logische Shiftoperatoren

➥ Bits werden im Wort verschoben

<<	Links-Shift
>>	Rechts-Shift

■ Beispiel:

x	1	0	0	1	1	1	0	0
x << 2	0	1	1	1	0	0	0	0

7 Inkrement / Dekrement Operatoren

`++`	inkrement
`--`	dekrement

- **linksseitiger Operator:** `++x` bzw. `--x`
  - es wird der Inhalt von `x` inkrementiert bzw. dekrementiert
  - das Resultat wird als Ergebnis geliefert
- **rechtsseitiger Operator:** `x++` bzw. `x--`
  - es wird der Inhalt von `x` als Ergebnis geliefert
  - anschließend wird `x` inkrementiert bzw. dekrementiert.
- Beispiele:

```
a = 10;
b = a++;        /* -> b: 10 und a: 11 */
c = ++a;        /* -> c: 12 und a: 12 */
```


8 Bedingte Bewertung

A ? B : C

- der Operator dient zur Formulierung von Bedingungen in Ausdrücken
- zuerst wird Ausdruck **A** bewertet
- ist **A ungleich 0**, so hat der gesamte Ausdruck als Wert den Wert des Ausdrucks **B**,
- sonst den Wert des Ausdrucks **C**
- Beispiel:

```
c = a>b ? a : b;            /* z = max(a,b) */
```

besser:

```
c = (a>b) ? a : b;
```

9 Komma-Operator

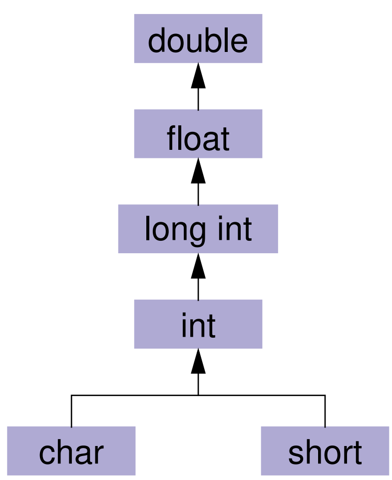
```
,
```

➥ der Komma-Operator erlaubt die Aneinanderreihung mehrerer Ausdrücke

- ein so gebildeter Ausdruck hat als Wert den Wert des letzten Teil-Ausdrucks

10 Typumwandlung in Ausdrücken

- Enthält ein Ausdruck Operanden unterschiedlichen Typs, erfolgt eine automatische Umwandlung in den Typ des in der **Hierarchie der Typen** am höchsten stehenden Operanden. (*Arithmetische Umwandlungen)*



Hierarchie der Typen (Auszug)

11 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

D.6 Einfacher Programmaufbau

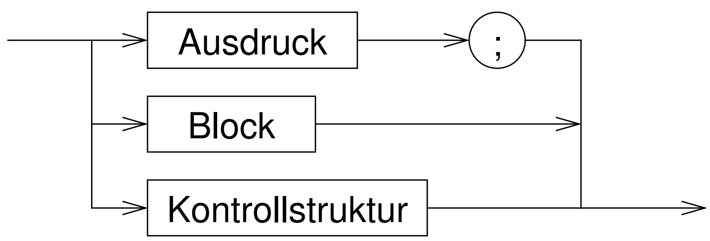
- Struktur eines C-Hauptprogramms
- Anweisungen und Blöcke
- Einfache Ein-/Ausgabe
- C-Präprozessor

1 Struktur eines C-Hauptprogramms

```
main()
{
        Variablendefinitionen
        Anweisungen
}
```

2 Anweisungen

Anweisung:



3 Blöcke

- Zusammenfassung mehrerer Anweisungen
- Lokale Variablendefinitionen → Hilfsvariablen
- Schaffung neuer Sichtbarkeitsbereiche (***Scopes***) für Variablen
  - ◆ bei Namensgleichheit ist immer die Variable des innersten Blocks sichtbar

```
main()
{
    int x, y, z;
    x = 1;
    {
        int a, b, c;
        a = x+1;
        {
            int a, x;
            x = 2;
            a = 3;
        }
        /* a: 2, x: 1 */
    }
}
```

4 Einfache Ein-/Ausgabe

- Jeder Prozess (jedes laufende Programm) bekommt von der Shell als Voreinstellung drei Ein-/Ausgabekanäle:

stdin	als Standardeingabe
stdout	als Standardausgabe
stderr	Fehlerausgabe

- Die Kanäle **stdin**, **stdout** und **stderr** sind in UNIX auf der Kommandozeile umlenkbar:

```
% prog < EingabeDatei > AusgabeDatei
```


4 Einfache Ein-/Ausgabe (2)

- Für die Sprache C existieren folgende primitive Ein-/Ausgabefunktionen für die Kanäle **stdin** und **stdout**:

getchar	zeichenweise Eingabe
putchar	zeichenweise Ausgabe
scanf	formatierte Eingabe
printf	formatierte Ausgabe

- folgende Funktionen ermöglichen Ein-/Ausgabe auf beliebige Kanäle (z. B. auch **stderr**)

  getc, **putc**, **fscanf**, **fprintf**

5 Einzelzeichen E/A

- ***getchar( ), getc( )*** ein Zeichen lesen
  - Beispiel:

```
int c;
c = getchar();
```

```
int c;
c = getc(stdin);
```

- ***putchar( ), putc( )*** ein Zeichen schreiben
  - Beispiel:

```
char c = 'a';
putchar(c);
```

```
char c = 'a';
putc(c, stdout);
```

- Beispiel:

```
#include <stdio.h>

/*
 * kopiere Eingabe auf Ausgabe
 */
main()
{
    int c;
    while ( (c = getchar()) != EOF )
    {
        putchar(c);
    }
}
```


6 Formatierte Ausgabe

- Aufruf: `printf ( `*format*`, `*arg*`)`
- ***printf*** konvertiert, formatiert und gibt die **Werte *(arg)*** unter der Kontrolle des Formatstrings ***format*** aus
  - die Anzahl der Werte (*arg*) ist abhängig vom Formatstring
- sowohl für ***format***, wie für ***arg*** sind Ausdrücke zulässig
- ***format*** ist vom Typ **Zeichenkette *(string)***
- ***arg*** muss dem durch das zugehörige **Formatelement** beschriebenen Typ entsprechen