Systemnahe Programmierung in C

6 Einfache Datentypen

J. Kleinöder, D. Lohmann, V. Sieh

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2024

http://sys.cs.fau.de/lehre/ss24

Was ist ein Datentyp?

- **Datentyp** := (<*Menge von Werten*>, <*Menge von Operationen*>)
  - Literal — Wert im Quelltext
  - Konstante — Bezeichner für einen Wert
  - Variable — Bezeichner für Speicherplatz, der einen Wert aufnehmen kann
  - Funktion — Bezeichner für Sequenz von Anweisungen, die einen Wert zurückgibt

↪ 5–6

⇝ Literale, Konstanten, Variablen, Funktionen haben einen **(Daten-)Typ**

- Datentyp legt fest
  - Repräsentation der Werte im Speicher
  - Größe des Speicherplatzes für Variablen
  - Erlaubte Operationen
- Datentyp wird festgelegt
  - Explizit, durch Deklaration, Typ-Cast oder Schreibweise (Literale)
  - Implizit, durch „Auslassung“ (⇝ int schlechter Stil!)

Primitive Datentypen in C

- Ganzzahlen/Zeichen `char`, `short`, `int`, `long`, `long long` (C99)
  - Wertebereich: implementierungsabhängig [≠Java]
    Es gilt: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long`
  - Jeweils als `signed`- und `unsigned`-Variante verfügbar
- Fließkommazahlen `float`, `double`, `long double`
  - Wertebereich: implementierungsabhängig [≠Java]
    Es gilt: `float` ≤ `double` ≤ `long double`
  - Ab C99 auch als `_Complex`-Datentypen verfügbar (für komplexe Zahlen)
- Leerer Datentyp `void`
  - Wertebereich: $\emptyset$
- Boolescher Datentyp `_Bool` (C99)
  - Wertebereich: $\{0, 1\}$ ($\hookleftarrow$ letztlich ein Integertyp)
  - Bedingungsausdrücke (z. B. `if(...)`) sind in C vom Typ `int`! [≠Java]

Integertypen [≈Java]

- Integertyp — Verwendung — Literalformen
  - `char` — kleine Ganzzahl oder Zeichen — `'A'`, `65`, `0x41`, `0101`
  - `short [int]` — Ganzzahl (int ist optional) — s. o.
  - `int` — Ganzzahl „natürlicher Größe“ — s. o.
  - `long [int]` — große Ganzzahl — `65L`, `0x41L`, `0101L`
  - `long long [int]` — sehr große Ganzzahl — `65LL`, `0x41LL`, `0101LL`
- Typ-Modifizierer — werden vorangestellt — Literal-Suffix
  - signed — Typ ist vorzeichenbehaftet (Normalfall) — -
  - unsigned — Typ ist vorzeichenlos — `U`
  - const — Variable des Typs kann nicht verändert werden — -
- Beispiele (Variablendefinitionen)

```
char a                = 'A';    // char-Variable, Wert 65 (ASCII: A)
const int b           = 0x41;   // int-Konstante, Wert 65 (Hex: 0x41)
long c                = 0L;     // long-Variable, Wert 0
unsigned long int d = 22UL;   // unsigned-long-Variable, Wert 22
```


Integertypen: Größe und Wertebereich [≠Java]

- Die interne Darstellung (Bitbreite) ist implementierungsabhängig

	Datentyp-Breite in Bit				
	Java	C-Standard	gcc_{IA32}	gcc_{IA64}	gcc_{AVR}
`char`	16	≥ 8	8	8	8
`short`	16	≥ 16	16	16	16
`int`	32	≥ 16	32	32	16
`long`	64	≥ 32	32	64	32
`long long`	-	≥ 64	64	64	64

- Der Wertebereich berechnet sich aus der Bitbreite
  - signed $\quad -(2^{Bits-1}-1) \longrightarrow +(2^{Bits-1}-1)$
  - unsigned $\quad 0 \longrightarrow +(2^{Bits}-1)$

Integertypen: Größe und Wertebereich [≠Java]

- Die interne Darstellung (Bitbreite) ist implementierungsabhängig

	Datentyp-Breite in Bit				
	Java	C-Standard	gcc_{IA32}	gcc_{IA64}	gcc_{AVR}
`char`	16	≥ 8	8	8	8
`short`	16	≥ 16	16	16	16
`int`	32	≥ 16	32	32	16
`long`	64	≥ 32	32	64	32
`long long`	-	≥ 64	64	64	64

- Der Wertebereich berechnet sich aus der Bitbreite
  - signed $\quad -(2^{Bits-1}-1) \longrightarrow +(2^{Bits-1}-1)$
  - unsigned $\quad 0 \longrightarrow +(2^{Bits}-1)$

Hier zeigt sich die C-Philosophie: Effizienz durch **Maschinennähe** ↪ 3–17

Die interne Repräsentation der Integertypen ist definiert durch die **Hardware** (Registerbreite, Busbreite, etc.). Das führt im Ergebnis zu **effizientem Code**.

Integertypen: Maschinennähe $\longrightarrow$ Problemnähe

- **Problem:** Breite ($\rightsquigarrow$ Wertebereich) der C-Standardtypen ist implementierungsspezifisch $\mapsto$ **Maschinennähe**
- **Oft benötigt:** Integertyp definierter Größe $\mapsto$ **Problemnähe**
  - Wertebereich sicher, aber möglichst kompakt darstellen
  - Register definierter Breite n bearbeiten
  - Code unabhängig von Compiler und Hardware halten ($\rightsquigarrow$ Portierbarkeit)

Integertypen: Maschinennähe $\longrightarrow$ Problemnähe

- **Problem:** Breite ($\rightsquigarrow$ Wertebereich) der C-Standardtypen ist implementierungsspezifisch $\mapsto$ **Maschinennähe**
- **Oft benötigt:** Integertyp definierter Größe $\mapsto$ **Problemnähe**
  - Wertebereich sicher, aber möglichst kompakt darstellen
  - Register definierter Breite n bearbeiten
  - Code unabhängig von Compiler und Hardware halten ($\rightsquigarrow$ Portierbarkeit)
- **Lösung:** Modul `stdint.h`
  - Definiert Alias-Typen: `int`n`_t` und `uint`n`_t` für $n \in \{8, 16, 32, 64\}$
  - Wird vom Compiler-Hersteller bereitgestellt

Integertypen: Maschinennähe $\longrightarrow$ Problemnähe

- **Problem:** Breite ($\rightsquigarrow$ Wertebereich) der C-Standardtypen ist implementierungsspezifisch $\mapsto$ **Maschinennähe**
- **Oft benötigt:** Integertyp definierter Größe $\mapsto$ **Problemnähe**
  - Wertebereich sicher, aber möglichst kompakt darstellen
  - Register definierter Breite n bearbeiten
  - Code unabhängig von Compiler und Hardware halten ($\rightsquigarrow$ Portierbarkeit)
- **Lösung:** Modul `stdint.h`
  - Definiert Alias-Typen: `int`n`_t` und `uint`n`_t` für $n \in \{8, 16, 32, 64\}$
  - Wird vom Compiler-Hersteller bereitgestellt

Wertebereich `stdint.h`-Typen							
`uint8_t`	0	$\rightarrow$	255	`int8_t`	-128	$\rightarrow$	$+127$
`uint16_t`	0	$\rightarrow$	65.535	`int16_t`	-32.768	$\rightarrow$	$+32.767$
`uint32_t`	0	$\rightarrow$	4.294.967.295	`int32_t`	$-2.147.483.648$	$\rightarrow$	$+2.147.483.647$
`uint64_t`	0	$\rightarrow$	$> 1,8 * 10^{19}$	`int64_t`	$< -9,2 * 10^{18}$	$\rightarrow$	$> +9,2 * 10^{18}$

Integertypen: Maschinennähe $\longrightarrow$ Problemnähe

- **Problem:** Breite ($\rightsquigarrow$ Wertebereich) der C-Standardtypen ist implementierungsspezifisch $\mapsto$ **Maschinennähe**
- **Oft benötigt:** Integertyp definierter Größe $\mapsto$ **Problemnähe**
  - Wertebereich sicher, aber möglichst kompakt darstellen
  - Register definierter Breite *n* bearbeiten
  - Code unabhängig von Compiler und Hardware halten ($\rightsquigarrow$ Portierbarkeit)
- **Lösung:** Modul `stdint.h`
  - Definiert Alias-Typen: `int`*n*`_t` und `uint`*n*`_t`
  - Wird vom Compiler-Hersteller bereitgestellt

Wie definiert man **problemspezifische** Typen?

Wertebereich `stdint.h`-Typen							
`uint8_t`	0	$\rightarrow$	255	`int8_t`	-128	$\rightarrow$	$+127$
`uint16_t`	0	$\rightarrow$	65.535	`int16_t`	-32.768	$\rightarrow$	$+32.767$
`uint32_t`	0	$\rightarrow$	4.294.967.295	`int32_t`	$-2.147.483.648$	$\rightarrow$	$+2.147.483.647$
`uint64_t`	0	$\rightarrow$	$> 1,8 * 10^{19}$	`int64_t`	$< -9,2 * 10^{18}$	$\rightarrow$	$> +9,2 * 10^{18}$

Typ-Aliase mit `typedef` [≠Java]

- Mit dem `typedef`-Schlüsselwort definiert man einen Typ-Alias:

  `typedef` *Typausdruck* *Bezeichner*`;`

  - *Bezeichner* ist nun ein alternativer Name für *Typausdruck*
  - Kann überall verwendet werden, wo ein Typausdruck erwartet wird

```
// stdint.h (avr-gcc)                  // stdint.h (x86-gcc, IA32)
typedef unsigned char uint8_t;         typedef unsigned char  uint8_t;
typedef unsigned int  uint16_t;        typedef unsigned short uint16_t;
...                                    ...
```

```
// main.c
#include <stdint.h>

uint16_t counter = 0;     // global 16-bit counter, range 0-65535
...
typedef uint8_t Register; // Registers on this machine are 8-bit
...
```


Typ-Aliase mit `typedef` (Forts.) [≠Java]

- Typ-Aliase ermöglichen einfache problembezogene Abstraktionen
  - `Register` ist problemnäher als `uint8_t`
    ⇝ Spätere Änderungen (z. B. auf 16-Bit-Register) zentral möglich
  - `uint16_t` ist problemnäher als `unsigned char`
  - `uint16_t` ist sicherer als `unsigned char`

Definierte Bitbreiten sind bei der μC-Entwicklung sehr wichtig!

- Große Unterschiede zwischen Plattformen und Compilern
  ⇝ Kompatibilitätsprobleme
- Um Speicher zu sparen, sollte immer der **kleinstmögliche** Integertyp verwendet werden

Regel: Bei der systemnahen Programmierung werden Typen aus `stdint.h` verwendet!

Aufzählungstypen mit `enum` [≈Java]

- Mit dem `enum`-Schlüsselwort definiert man einen Aufzählungstyp über eine explizite Menge symbolischer Werte:

  `enum` *Bezeichner*$_{opt}$ `{` *KonstantenListe* `} ;`

- Beispiel
  - Definition:

```
enum eLED {RED0, YELLOW0, GREEN0, BLUE0,
           RED1, YELLOW1, GREEN1, BLUE1};
```

  - Verwendung:

```
enum eLED myLed = YELLOW0;  // enum necessary here!
...
sb_led_on(BLUE1);
```

Aufzählungstypen mit `enum` [≈Java]

- Mit dem `enum`-Schlüsselwort definiert man einen Aufzählungstyp über eine explizite Menge symbolischer Werte:

  `enum` *Bezeichner*$_{opt}$ `{` *KonstantenListe* `} ;`

- Beispiel
  - Definition:

```
enum eLED {RED0, YELLOW0, GREEN0, BLUE0,
           RED1, YELLOW1, GREEN1, BLUE1};
```

  - Verwendung:

```
enum eLED myLed = YELLOW0;  // enum necessary here!
...
sb_led_on(BLUE1);
```

- Vereinfachung der Verwendung durch `typedef`
  - Definition:

```
typedef enum eLED {RED0, YELLOW0, GREEN0, BLUE0,
                   RED1, YELLOW1, GREEN1, BLUE1} LED;
```

  - Verwendung:

```
LED myLed = YELLOW0;  // LED --> enum eLED
```


enum ↦ int [≠Java]

- Technisch sind enum-Typen Integers (int)
  - enum-Konstanten werden von 0 an durchnummeriert

```
typedef enum { RED0,      // value: 0
               YELLOW0,   // value: 1
               GREEN0,    // value: 2
               ··· } LED;
```

  - Es ist auch möglich, Werte direkt zuzuweisen

```
typedef enum { BUTTON0 = 4, BUTTON1 = 8 } BUTTON;
```

  - Man kann sie verwenden wie ints (z. B. mit ihnen rechnen)

```
sb_led_on(RED0 + 2);  // -> LED GREEN0 is on
sb_led_on(1);         // -> LED YELLOW0 is on
for (int led = RED0; led <= BLUE1; led++)
  sb_led_off(led);    // turn off all LEDs
// Also possible...
sb_led_on(4711);      // no compiler/runtime error!
```

- ⇝ Es findet **keinerlei Typprüfung** statt!

enum ↦ int [≠Java]

- Technisch sind enum-Typen Integers (int)
  - enum-Konstanten werden von 0 an durchnummeriert

```
typedef enum { RED0,      // value: 0
               YELLOW0,   // value: 1
               GREEN0,    // value: 2
               ··· } LED;
```

  - Es ist auch möglich, Werte direkt zuzuweisen

```
typedef enum { BUTTON0 = 4, BUTTON1 = 8 } BUTTON;
```

  - Man kann sie verwenden wie ints (z. B. mit ihnen rechnen)

```
sb_led_on(RED0 + 2);  // -> LED GREEN0 is on
sb_led_on(1);         // -> LED YELLOW0 is on
for (int led = RED0; led <= BLUE1; led++)
  sb_led_off(led);    // turn off all LEDs
// Also possible...
sb_led_on(4711);      // no compiler/runtime error!
```

- ⇝ Es findet **keinerlei Typprüfung** statt!

Das entspricht der **C-Philosophie**! ↪ 3–17

Fließkommatypen [≈Java]

- Fließkommatyp — Verwendung — Literalformen
  - `float` — einfache Genauigkeit (≈ 7 St.) — `100.0F`, `1.0E2F`
  - `double` — doppelte Genauigkeit (≈ 15 St.) — `100.0`, `1.0E2`
  - `long double` — „erweiterte Genauigkeit“ — `100.0L` `1.0E2L`
- Genauigkeit / Wertebereich sind implementierungsabhängig [≠Java]
  - Es gilt: `float` ≤ `double` ≤ `long double`
  - `long double` und `double` sind auf vielen Plattformen identisch

„Effizienz durch Maschinennähe“ ↪ 3–17

Fließkommatypen [≈Java]

- Fließkommatyp — Verwendung — Literalformen
  - `float` — einfache Genauigkeit (≈ 7 St.) — `100.0F`, `1.0E2F`
  - `double` — doppelte Genauigkeit (≈ 15 St.) — `100.0`, `1.0E2`
  - `long double` — „erweiterte Genauigkeit“ — `100.0L` `1.0E2L`
- Genauigkeit / Wertebereich sind implementierungsabhängig [≠Java]
  - Es gilt: `float` ≤ `double` ≤ `long double`
  - `long double` und `double` sind auf vielen Plattformen identisch

„Effizienz durch Maschinennähe“ ↪ 3–17

Fließkommazahlen + μC-Plattform = $$$

- Oft keine Hardwareunterstützung für `float`-Arithmetik
  ⇝ **sehr teure** Emulation in Software (langsam, viel zusätzlicher Code)
- Speicherverbrauch von `float`- und `double`-Variablen ist **sehr hoch**
  ⇝ mindestens 32/64 Bit (`float`/`double`)

Regel: Bei der μ-Controller-Programmierung ist auf Fließkommaarithmetik **zu verzichten**!

Zeichen $\mapsto$ Integer [$\approx$Java]

- Zeichen sind in C ebenfalls Ganzzahlen (Integers) $\hookrightarrow$ 6–3
  - `char` gehört zu den Integer-Typen (üblicherweise 8 Bit = 1 Byte)
- Repräsentation erfolgt durch den ASCII-Code $\hookrightarrow$ 6–12
  - 7-Bit-Code $\mapsto$ 128 Zeichen standardisiert
    (die verbleibenden 128 Zeichen werden unterschiedlich interpretiert)
  - Spezielle Literalform durch Hochkommata
    `'A'` $\mapsto$ ASCII-Code von `A`
  - Nichtdruckbare Zeichen durch Escape-Sequenzen
    - Tabulator `'\t'`
    - Zeilentrenner `'\n'`
    - Backslash `'\\'`
- Zeichen $\mapsto$ Integer $\rightsquigarrow$ man kann mit Zeichen rechnen

```
char b = 'A' + 1;          // b: 'B'

int lower(int ch) {        // lower('X'): 'x'
  return ch + 0x20;
}
```

ASCII-Code-Tabelle (7 Bit)

ASCII $\mapsto$ *American Standard Code for Information Interchange*

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** 4B	**L** 4C	**M** 4D	**N** 4E	**O** 4F
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

Zeichenketten (Strings) [≠Java]

- Ein String ist in C ein Feld (Array) von Zeichen
  - Repräsentation: Folge von Einzelzeichen, terminiert durch (letztes Zeichen): `NUL` (ASCII-Wert 0)
  - Speicherbedarf: (Länge + 1) Bytes
- Spezielle Literalform durch doppelte Hochkommata:

- Beispiel (Linux)

```
#include <stdio.h>

char string[] = "Hello, World!\n";

int main(void) {
  printf("%s", string);
  return 0;
}
```

Zeichenketten (Strings) [≠Java]

- Ein String ist in C ein Feld (Array) von Zeichen
  - Repräsentation: Folge von Einzelzeichen, terminiert durch (letztes Zeichen): `NUL` (ASCII-Wert 0)
  - Speicherbedarf: (Länge + 1) Bytes
- Spezielle Literalform durch doppelte Hochkommata:

`"Hi!"` $\mapsto$

`'H'`	`'i'`	`'!'`	`0`

$\leftarrow$ abschließendes 0-Byte

- Beispiel (Linux)

```
#include <stdio.h>

char string[] = "Hello, World!\n";

int main(void) {
  printf("%s", string);
  return 0;
}
```

Zeichenketten brauchen vergleichsweise viel Speicher und „größere“ Ausgabegeräte (z. B. LCD-Display).
$\rightsquigarrow$ Bei der μC-Programmierung spielen sie nur eine untergeordnete Rolle.

Ausblick: Komplexe Datentypen

- Aus einfachen Datentypen lassen sich (rekursiv) auch komplexe(re) Datentypen bilden
  - Felder (Arrays) $\hookrightarrow$ Sequenz von Elementen gleichen Typs [≈Java]

```
int intArray[4];          // allocate array with 4 elements
intArray[0] = 0x4711;     // set 1st element (index 0)
```

  - Zeiger $\hookrightarrow$ veränderbare Referenzen auf Variablen [≠Java]

```
int  a = 0x4711;          // a: 0x4711
int *b = &a;              // b: -->a (memory location of a)
int  c = *b;              // pointer dereference (c: 0x4711)
*b     = 23;              // pointer dereference (a: 23)
```

  - Strukturen $\hookrightarrow$ Verbund von Elementen bel. Typs [≠Java]

```
struct Point { int x; int y; };
struct Point p;           // p is Point variable
p.x = 0x47;               // set x-component
p.y = 0x11;               // set y-component
```

- Wir betrachten diese detailliert in späteren Kapiteln