System-Level Programming

6 Basic Data Types

Peter Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Summer Term 2025

http://sys.cs.fau.de/lehre/ss25

What Exactly is a Data Type?

- **Data type** := (*<set of values>*, *<set of operations>*)
  - Literal   value in the source code
  - Constant   identifier for a value
  - Variable   identifier for a memory address,
    where a value can be stored
  - Function   identifier for a sequence of instructions,
    which will return a value
  - ⤳ literals, constants, variables, functions all have a **(data) type**

- The data type determines
  - the representation of the value in memory
  - the **size** which gets occupied by the variable in storage
  - which **operations** are permitted

- The data type gets determined
  - explicitly, by declaration, type cast, or notation (literals)
  - implicitly, by "omitting" (⤳ int   bad style!)

Primitive Data Types in C

- Integers/characters   `char`, `short`, `int`, `long`, `long long` (C99)
  - range of values: dependent on implementation   [$\neq$Java]
    still: `char` $\leq$ `short` $\leq$ `int` $\leq$ `long` $\leq$ `long long`
  - both available in `signed` or `unsigned` version

- Floating-point numbers `float`, `double`, `long double`
  - range of values: dependent on implementation   [$\neq$Java]
    still: `float` $\leq$ `double` $\leq$ `long double`
  - From C99 onwards, they are available as `_Complex` data types (for complex numbers).

- Empty data type   `void`
  - range of values: $\emptyset$

- Boolean   `_Bool` (C99)
  - range of values: $\{0, 1\}$ ($\hookleftarrow$ actually only an integer type)
  - conditional expressions (e. g., `if(...)`) are of type `int`!   [$\neq$Java]

Integer Types [≈Java]

- Integer type   usage   literal from

Integer type	usage	literal from
char	small integer or character	'A', 65, 0x41, 0101
short [int]	integer (int is optional)	s. a.
int	integer of "natural size"	s. a.
long [int]	big integer	65L, 0x41L, 0101L
long long [int]	really big integer	65LL, 0x41LL, 0101LL

- Type modifier   get prefixed   literal suffix

Type modifier	get prefixed	literal suffix
signed	type is signed (standard case)	-
unsigned	type does not have a sign	U
const	variable cannot be changed	-

- Examples (definition of variables)

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

06-Datentypen_en

Integer Types: Size and Range of Values   [$\neq$Java]

■ The internal representation (width in bits) is dependent on implementation

	width of data types 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

■ The range of values can be calculated from the width in bits
  - signed $-(2^{bits-1}-1) \longrightarrow +(2^{bits-1}-1)$
  - unsigned $0 \longrightarrow +(2^{bits}-1)$

06-Datentypen_en

Integer Types: Size and Range of Values   [$\neq$Java]

- The internal representation (width in bits) is dependent on implementation

	Java	C Standard	gcc$_{IA32}$	gcc$_{IA64}$	gcc$_{AVR}$
		width of data types in bit			
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

- The range of values can be calculated from the width in bits
  - signed $\qquad\qquad -(2^{bits-1}-1) \quad \longrightarrow \quad +(2^{bits-1}-1)$
  - unsigned $\qquad\qquad\qquad 0 \quad \longrightarrow \quad +(2^{bits}-1)$

  The philosophy of C is obvious: Efficiency by **machine orientation**

  Internal representation of integer types is defined by the **hardware** (width of registers, bus, etc.). This yields code that is in general **more efficient**.

06-Datentypen _ en

■ **Problem:**   width ($\rightsquigarrow$ range of values) of C standard types is
              dependent on implementation
              $\mapsto$ **machine orientation**

■ **Often needed:** Integer types of specific size
              $\mapsto$ **problem orientation**

  ■ represent range of values safely, but as memory efficient as possible
  ■ dealing with registers of defined width n
  ■ keeping code independent of compiler and hardware ($\rightsquigarrow$ portability)

- **Problem:** width ($\rightsquigarrow$ range of values) of C standard types is dependent on implementation

  $\mapsto$ **machine orientation**

- **Often needed:** Integer types of specific size

  $\mapsto$ **problem orientation**
  - represent range of values safely, but as memory efficient as possible
  - dealing with registers of defined width n
  - keeping code independent of compiler and hardware ($\rightsquigarrow$ portability)

- **Solution:** module `stdint.h`
  - defines alias types: `int`n`_t` and `uint`n`_t`   for $n \in \{8, 16, 32, 64\}$
  - gets provided by compiler developers

Integer Types: Machine Orientation $\longrightarrow$ Problem Orientation

- **Problem:** width ($\leadsto$ range of values) of C standard types is dependent on implementation

  $\mapsto$ **machine orientation**
- **Often needed:** Integer types of specific size

  $\mapsto$ **problem orientation**
  - represent range of values safely, but as memory efficient as possible
  - dealing with registers of defined width n
  - keeping code independent of compiler and hardware ($\leadsto$ portability)
- **Solution:** module stdint.h
  - defines alias types: int$n$_t and uint$n$_t   for $n \in \{8, 16, 32, 64\}$
  - gets provided by compiler developers

range of values for stdint.h-types						
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	$< -9.2 * 10^{18} \rightarrow > +9.2 * 10^{18}$	

06-Datentypen_en

- **Problem:** width ($\rightsquigarrow$ range of values) of C standard types is dependent on implementation
  $\mapsto$ **machine orientation**
- **Often needed:** Integer types of specific size
  $\mapsto$ **problem orientation**
  - represent range of values safely, but as memory efficient as possible
  - dealing with registers of defined width n
  - keeping code independent of compiler and hardware ($\rightsquigarrow$ portability)
- **Solution:** module stdint.h
  - defines alias types: int$n$_t and uint$n$_t
  - gets provided by compiler developers

How to define **problem-specific** types?

range of values for stdint.h-types						
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	$< -9.2 * 10^{18} \rightarrow > +9.2 * 10^{18}$	

06-Datentypen_en

Type Aliases with `typedef` [$\neq$ Java]

- With help of the keyword `typedef`, possibility to define a type alias:

  `typedef` *alias identifier* ;

  - *identifier* is now an alternative name for a *type expression*
  - It can be used at any place a type expression is expected.

```
// 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
...
```

- Type aliases enable easy problem-specific abstractions
  - `register` is closer to the problem than `uint8_t`
    $\rightsquigarrow$ later (e. g., with 16-bit-registers) modification possible
  - `uint16_t` is closer to the problem than `unsigned char`
  - `uint16_t` is safer than `unsigned char`

    > Definied bit widths are crucial for μC development!
    > - Major differences between platforms and compilers
    > $\rightsquigarrow$ compatibility problems
    > - To save memory, the **smallest possible** integer type should always be used!
    >
    > **Rule:**   For system-level programming types
    >   from `stdint.h` get used!

Enumeration Types with enum

■ With help of the keyword **enum**, an enumeration type is defined, consisting of an explicit set of symbolic values:

  enum *identifier_opt* { *listofconstants* } ;

■ Example

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

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

Enumeration Types with `enum`

- With help of the keyword **enum**, an enumeration type is defined, consisting of an explicit set of symbolic values:

  enum *identifier$_{opt}$* { *listofconstants* } ;

- Example

  - definition:

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

  - usage:

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

- Simplification with `typedef`

  - definition:

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

  - usage:

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

06-Datentypen_en

- enum types are technically nothing else than integers (int)
  - enum constants get enumerated, starting from 0

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

  - possibility to explicitly assign values:

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

  - they can be used like ints (e. g., arithmetic operations)

    ```c
    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!
    ```

- ↝ There will be **no type checks**!

06-Datentypen_en

enum ↦ int   [≠ Java]

- enum types are technically nothing else than integers (int)
  - enum constants get enumerated, starting from 0

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

  - possibility to explicitly assign values:

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

  - they can be used like ints (e. g., arithmetic operations)

    ```
    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!
    ```

- ⤳ There will be **no type checks**!   This conforms to
                                         C philosophy!

06-Datentypen_en

Floating-Point (FP) Types [≈Java]

- FP type   usage   literal form
  - `float`   single precision   `100.0F`, `1.0E2F`
  - `double`   double precision   `100.0`, `1.0E2`
  - `long double`   "extended precision"   `100.0L 1.0E2L`

- Precision / range of values are implementation-dependent [≠Java]
  - still: `float` ≤ `double` ≤ `long double`
  - `long double` and `double` are identical on most platforms

  > "efficiency by machine orientation"

Floating-Point (FP) Types [≈Java]

- FP type   usage   literal form
  - `float`   single precision   `100.0F`, `1.0E2F`
  - `double`   double precision   `100.0`, `1.0E2`
  - `long double`   "extended precision"   `100.0L` `1.0E2L`

- Precision / range of values are implementation-dependent   [≠Java]
  - still: `float` ≤ `double` ≤ `long double`
  - `long double` and `double` are identical on most platforms

> "efficiency by machine orientation"

> **Floats + µC platform = $$$**
> - Often, µCs have no native hardware support for `float` arithmetic.
>   ↝ **really expensive** emulation in software (slow, much additional code)
> - Memory demand of `float`- and `double` variables is **really high**
>   ↝ at least 32/64 bit (`float`/`double`)
>
> **Rule:**   When programming a µController, floating-point arithmetic **should be avoided**!

- In C, characters are integers　　　　　　　　　　$\hookrightarrow$ 6–3
  - `char` is part of the integer types (usually 8 bit = 1 byte)
- Representation takes place with ASCII code　　　$\hookrightarrow$ 6–12
  - 7-bit code $\mapsto$ 128 standardized characters
    (the remaining 128 characters can be interpreted differently)
  - special literal form with single quote marks
    `'A'` $\mapsto$ ASCII code of `A`
  - non-printable characters with escape sequences
    - tabulator　　　　　　`'\t'`
    - line separator　　　　`'\n'`
    - backslash　　　　　　`'\\'`
- character $\mapsto$ integer　　$\leadsto$　　characters can be used in operations

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

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

06-Datentypen_en

ASCII-Code Table (7 bit)

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

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

Strings   [≠Java]

- In C, a string is an array of characters.
  - representation: sequence of single characters, terminated by
                        (last character): NUL (ASCII value 0)
  - memory demand: (length + 1) bytes
- Special literal form with double quotes:

  "Hi!" ↦ | 'H' | 'i' | '!' | **0** ←—— terminating 0 byte

- Example (Linux)

```
#include <stdio.h>

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

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

Strings [≠Java]

- In C, a string is an array of characters.
  - representation: sequence of single characters, terminated by (last character): `NUL` (ASCII value 0)
  - memory demand: (length + 1) bytes
- Special literal form with double quotes:

  `"Hi!"` ↦ | `'H'` | `'i'` | `'!'` | **0** ← — terminating 0 byte

- Example (Linux)

```c
#include <stdio.h>

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

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

> Strings need relatively much memory and "larger" output devices (e. g., LCD display).
> ↝ For µC programming they only have a minor significance.

06-Datentypen_en

Outlook: Complex Data Types

- From primitive data types, complex date types can be created (recursively)

  - Arrays   ↪ element sequence (same type)   [≈Java/Python]

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

  - Pointers   ↪ modifiable reference to a variable   [≠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)
    ```

  - Structures   ↪ composition of elements of any type   [≠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
    ```

- We have a closer look at this in later chapters.

06-Datentypen_en