System-Level Programming

6 Basic Data Types

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2024

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

What Exactly is a Data Type?

- **Data type** := (<*set of values*>, <*set of operations*>)
  - Literal — value in the source ↪ ??
  - 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 [≠Java]
    still: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long`
  - both available in `signed` or `unsigned` version
- Floating-point numbers `float`, `double`, `long double`
  - range of values: dependent on implementation [≠Java]
    still: `float` ≤ `double` ≤ `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`! [≠Java]

06-Datentypen_en

Integer Types [≈Java]

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
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
```

Integer Types: Size and Range of Values [≠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 [≠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 $0 \longrightarrow +(2^{bits}-1)$

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**.

Integer Types: Machine Orientation ⟶ Problem Orientation

- **Problem:** width (⇝ range of values) of C standard types is dependent on implementation
  ↦ **machine orientation**
- **Often needed:** Integer types of specific size
  ↦ **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 (⇝ portability)

Integer Types: Machine Orientation $\longrightarrow$ Problem Orientation

- **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

06-Datentypen_en

Integer Types: Machine Orientation $\longrightarrow$ Problem Orientation

- **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

range of values for `stdint.h`-types							
`uint8_t`	0	→	255	`int8_t`	−128	→	+127
`uint16_t`	0	→	65 535	`int16_t`	−32 768	→	+32 767
`uint32_t`	0	→	4 294 967 295	`int32_t`	−2 147 483 648	→	+2 147 483 647
`uint64_t`	0	→	$> 1.8 * 10^{19}$	`int64_t`	$< -9.2 * 10^{18}$	→	$> +9.2 * 10^{18}$

Integer Types: Machine Orientation ⟶ Problem Orientation

- **Problem:** width (⇝ range of values) of C standard types is dependent on implementation
  ↦ **machine orientation**
- **Often needed:** Integer types of specific size
  ↦ **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 (⇝ portability)
- **Solution:** module `stdint.h`
  - defines alias types: `int`*n*`_t` and `uint`*n*`_t`
  - gets provided by compiler developers

How can one define **problem-specific** types?

range of values for `stdint.h`-types							
`uint8_t`	0	→	255	`int8_t`	−128	→	+127
`uint16_t`	0	→	65 535	`int16_t`	−32 768	→	+32 767
`uint32_t`	0	→	4 294 967 295	`int32_t`	−2 147 483 648	→	+2 147 483 647
`uint64_t`	0	→	$> 1.8 * 10^{19}$	`int64_t`	$< -9.2 * 10^{18}$	→	$> +9.2 * 10^{18}$

Type Aliases with `typedef` [≠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.

```c
// 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;
...                                    ...
```

```c
// 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 with `typedef` (continued) [≠Java]

- Type aliases enable easy problem-specific abstractions
  - `register` is closer to the problem than `uint8_t`
    ⇝ 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
  ⇝ 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** [≈Java]

- 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** [≈Java]

- 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
    ```

`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 `int`s (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**!

06-Datentypen_en

enum ↦ int

- **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**!

Floating-Point (FP) Types [≈Java]

- FP type usage literal form

FP type	usage	literal form
`float`	single precision (≈ 7 St.)	`100.0F`, `1.0E2F`
`double`	double precision (≈ 15 St.)	`100.0`, `1.0E2`
`long double`	"extended precision"	`100.0L` `1.0E2L`

- Precision / range of values are implementation-dependent [≠Java]
  - still: `float` $\leq$ `double` $\leq$ `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

FP type	usage	literal form
`float`	single precision (≈ 7 St.)	`100.0F`, `1.0E2F`
`double`	double precision (≈ 15 St.)	`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, there is no hardware support for `float` arithmetic.
  ⇝ **really expensive** emulation in software (slow, lots of additional code)
- Memory usage of `float`- and `double` variables is **really high**
  ⇝ at least 32/64 bit (`float`/`double`)

Rule: When programming a μController, floating-point arithmetic **should not be used at all**!

- In C, characters are integers ↪ 6–3
  - char is part of the integer types (usually 8 bit = 1 byte)
- Representation takes place with ASCII code ↪ 6–18
  - 7-bit code ↦ 128 standardized characters
    (the remaining 128 characters can get interpreted differently)
  - special literal form with superscripts
    'A' ↦ ASCII code of A
  - non-printable characters with escape sequences
    - tabulator '\t'
    - line separator '\n'
    - backslash '\\'
- character ↦ integer ⇝ characters can be used in operations

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

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

ASCII-Code Table (7 bit)

ASCII ↦ *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

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 usage: (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 usage: (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 need relatively much memory and "bigger" output devices (e. g., LCD display).
⇝ For μC programming they only have a small significance.

06-Datentypen_en

Outlook: Complex Data Types

- From small data types, more complex data types can be created (recursively)
  - Arrays ↪ sequence of elements of same type [≈Java]

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

  - Pointers ↪ modifiable reference to a variable [≠Java]

```c
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]

```c
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.