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**: $(<\text{set of values}>, <\text{set of operations}>)$
  - 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
  - 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
  - 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: `{}`

- **Boolean** `_Bool` (C99)
  - range of values: `{0, 1}` (← actually only an integer type)
  - conditional expressions (e.g., `if(...)`) are of type `int`!
# Integer Types

<table>
<thead>
<tr>
<th>Integer type</th>
<th>usage</th>
<th>literal from</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>small integer or character</td>
<td>'A', 65, 0x41, 0101</td>
</tr>
<tr>
<td>short [int]</td>
<td>integer (int is optional)</td>
<td>s. a.</td>
</tr>
<tr>
<td>int</td>
<td>integer of “natural size”</td>
<td>s. a.</td>
</tr>
<tr>
<td>long [int]</td>
<td>big integer</td>
<td>65L, 0x41L, 0101L</td>
</tr>
<tr>
<td>long long [int]</td>
<td>really big integer</td>
<td>65LL, 0x41LL, 0101LL</td>
</tr>
</tbody>
</table>

- **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)**

```c
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
```
The internal representation (width in bits) is dependent on implementation.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>width of data types in bit</th>
<th>C Standard</th>
<th>gcc_A32</th>
<th>gcc_A64</th>
<th>gcc_AVR</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>16</td>
<td>≥ 8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>≥ 16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>≥ 16</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>≥ 32</td>
<td>32</td>
<td>64</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>long long</td>
<td>-</td>
<td>≥ 64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

The range of values can be calculated from the width in bits:

- **signed**: $\neg (2^{bits-1} - 1) \rightarrow + (2^{bits-1} - 1)$
- **unsigned**: $0 \rightarrow + (2^{bits} - 1)$
The internal representation (width in bits) is dependent on implementation.

<table>
<thead>
<tr>
<th>Type</th>
<th>Java</th>
<th>C Standard</th>
<th>gcc/A32</th>
<th>gcc/A64</th>
<th>gcc/AVR</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>16</td>
<td>≥ 8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>≥ 16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>≥ 16</td>
<td>32</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>≥ 32</td>
<td>32</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>long long</td>
<td>-</td>
<td>≥ 64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

The range of values can be calculated from the width in bits:

- **signed**
  \[-(2^{\text{bits}}-1 - 1) \rightarrow + (2^{\text{bits}}-1 - 1)\]
- **unsigned**
  \[0 \rightarrow + (2^{\text{bits}} - 1)\]

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**.
Problem: width (\(\sim\) 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 (\(\sim\) portability)
**Problem:** width (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 (\( \mapsto \) portability)

**Solution:** module \texttt{stdint.h}

- defines alias types: \texttt{int}_{n\_t} and \texttt{uint}_{n\_t} for \( n \in \{8, 16, 32, 64\} \)
- gets provided by compiler developers
### 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 **stdlib.h**
- defines alias types: `intn_t` and `uintn_t` for $n \in \{8, 16, 32, 64\}$
- gets provided by compiler developers

<table>
<thead>
<tr>
<th>Type</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint8_t</code></td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td><code>int8_t</code></td>
<td>-128</td>
<td>+127</td>
</tr>
<tr>
<td><code>uint16_t</code></td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td><code>int16_t</code></td>
<td>-32768</td>
<td>+32767</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td><code>int32_t</code></td>
<td>-2147483648</td>
<td>+2147483647</td>
</tr>
<tr>
<td><code>uint64_t</code></td>
<td>0</td>
<td>$&gt; 1.8 \times 10^{19}$</td>
</tr>
<tr>
<td><code>int64_t</code></td>
<td>$&lt; -9.2 \times 10^{18}$</td>
<td>$&gt; +9.2 \times 10^{18}$</td>
</tr>
</tbody>
</table>
**Problem:** width (\(\sim\) range of values) of C standard types is dependent on implementation
\(\rightarrow\) **machine orientation**

**Often needed:** Integer types of specific size
\(\rightarrow\) **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 (\(\sim\) portability)

**Solution:** module `stdint.h`
- defines alias types: `int\(n\)_t` and `uint\(n\)_t`
- gets provided by compiler developers

<table>
<thead>
<tr>
<th>data type</th>
<th>minimum value</th>
<th>maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint8_t</code></td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td><code>int8_t</code></td>
<td>-128</td>
<td>+127</td>
</tr>
<tr>
<td><code>uint16_t</code></td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td><code>int16_t</code></td>
<td>-32768</td>
<td>+32767</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td><code>int32_t</code></td>
<td>-2147483648</td>
<td>+2147483647</td>
</tr>
<tr>
<td><code>uint64_t</code></td>
<td>0</td>
<td>(\geq 1.8\times10^{19})</td>
</tr>
<tr>
<td><code>int64_t</code></td>
<td>(&lt; -9.2\times10^{18})</td>
<td>(\geq +9.2\times10^{18})</td>
</tr>
</tbody>
</table>

How can one define **problem-specific** types?
Type Aliases with `typedef`

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)
typedef unsigned char uint8_t;
typedef unsigned int uint16_t;
...  
// stdint.h (x86-gcc, IA32)
typedef unsigned char uint8_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
  - 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!
With help of the keyword `enum`, an enumeration type is defined, consisting of an explicit set of symbolic values:

```java
enum identifier_opt { listofconstants } ;
```

Example

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

- **usage:**
  ```java
  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:

```java
enum identifier_opt { listofconstants } ;
```

Example

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

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

Simplification with `typedef`

- **definition:**
  ```java
  typedef enum eLED {RED0, YELLOW0, GREEN0, BLUE0, RED1, YELLOW1, GREEN1, BLUE1} LED;
  ```

- **usage:**
  ```java
  LED myLed = YELLOW0;  // LED --&gt; enum eLED
  ```
**enum **→ **int**

### 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!**
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)

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

There will be **no type checks**!

This conforms to **C philosophy**!
## Floating-Point (FP) Types

<table>
<thead>
<tr>
<th>FP type</th>
<th>usage</th>
<th>literal form</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>single precision (≈ 7 St.)</td>
<td>100.0F, 1.0E2F</td>
</tr>
<tr>
<td>double</td>
<td>double precision (≈ 15 St.)</td>
<td>100.0, 1.0E2</td>
</tr>
<tr>
<td>long double</td>
<td>“extended precision“</td>
<td>100.0L 1.0E2L</td>
</tr>
</tbody>
</table>

- Precision / range of values are **implementation-dependent**  
  - still: float ≤ double ≤ long double
  - long double and double are identical on most platforms

> “efficiency by machine orientation“
## Floating-Point (FP) Types

### FP type
- **float**
  - single precision (≈ 7 St.)
  - literal form: `100.0F, 1.0E2F`
- **double**
  - double precision (≈ 15 St.)
  - literal form: `100.0, 1.0E2`
- **long double**
  - “extended precision“
  - literal form: `100.0L, 1.0E2L`

### Precision / range of values
- still: `float ≤ double ≤ long double`
- **long double** and **double** are identical on most platforms

### Implementation-dependent (≠ Java)
- “efficiency by machine orientation“

### Other Considerations
- Floats on 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
- `char` is part of the integer types (usually 8 bit = 1 byte)
representation takes place with ASCII code
- 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       \(\backslash t\)
  - line separator  \(\backslash n\)
  - backslash       \(\backslash\)

character → integer  →  characters can be used in operations

```c
char b = 'A' + 1;    // b: 'B'
int lower(int ch) {
  return ch + 0x20;
}
```
ASCII→American Standard Code for Information Interchange

<table>
<thead>
<tr>
<th>ASCII</th>
<th>(\rightarrow)</th>
<th>ASCII</th>
<th>(\rightarrow)</th>
<th>ASCII</th>
<th>(\rightarrow)</th>
<th>ASCII</th>
<th>(\rightarrow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>00</td>
<td>SOH</td>
<td>01</td>
<td>STX</td>
<td>02</td>
<td>ETX</td>
<td>03</td>
</tr>
<tr>
<td>BS</td>
<td>08</td>
<td>HT</td>
<td>09</td>
<td>NL</td>
<td>0A</td>
<td>VT</td>
<td>0B</td>
</tr>
<tr>
<td>DLE</td>
<td>10</td>
<td>DC1</td>
<td>11</td>
<td>DC2</td>
<td>12</td>
<td>DC3</td>
<td>13</td>
</tr>
<tr>
<td>CAN</td>
<td>18</td>
<td>EM</td>
<td>19</td>
<td>SUB</td>
<td>1A</td>
<td>ESC</td>
<td>1B</td>
</tr>
<tr>
<td>SP</td>
<td>20</td>
<td>!</td>
<td>21</td>
<td>&quot;</td>
<td>22</td>
<td>#</td>
<td>23</td>
</tr>
<tr>
<td>(</td>
<td>28</td>
<td>)</td>
<td>29</td>
<td>*</td>
<td>2A</td>
<td>+</td>
<td>2B</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>1</td>
<td>31</td>
<td>2</td>
<td>32</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>9</td>
<td>39</td>
<td>:</td>
<td>3A</td>
<td>;</td>
<td>3B</td>
</tr>
<tr>
<td>G</td>
<td>40</td>
<td>A</td>
<td>41</td>
<td>B</td>
<td>42</td>
<td>C</td>
<td>43</td>
</tr>
<tr>
<td>H</td>
<td>48</td>
<td>I</td>
<td>49</td>
<td>J</td>
<td>4A</td>
<td>K</td>
<td>4B</td>
</tr>
<tr>
<td>P</td>
<td>50</td>
<td>Q</td>
<td>51</td>
<td>R</td>
<td>52</td>
<td>S</td>
<td>53</td>
</tr>
<tr>
<td>X</td>
<td>58</td>
<td>Y</td>
<td>59</td>
<td>Z</td>
<td>5A</td>
<td>[</td>
<td>5B</td>
</tr>
<tr>
<td>\</td>
<td>60</td>
<td>a</td>
<td>61</td>
<td>b</td>
<td>62</td>
<td>c</td>
<td>63</td>
</tr>
<tr>
<td>h</td>
<td>68</td>
<td>i</td>
<td>69</td>
<td>j</td>
<td>6A</td>
<td>k</td>
<td>6B</td>
</tr>
<tr>
<td>p</td>
<td>70</td>
<td>q</td>
<td>71</td>
<td>r</td>
<td>72</td>
<td>s</td>
<td>73</td>
</tr>
<tr>
<td>x</td>
<td>78</td>
<td>y</td>
<td>79</td>
<td>z</td>
<td>7A</td>
<td>{</td>
<td>7B</td>
</tr>
</tbody>
</table>

© klsw System-Level Programming (ST 24) 6 Basic Data Types – Characters and Strings
Strings

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)

```c
#include <stdio.h>

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

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

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!" \(\mapsto\) 'H' 'i' '!' 0 \(\leftarrow\) 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 “bigger” output devices (e.g., LCD display).
\(~\) For \(\mu\)C programming they only have a small significance.
Outlook: Complex Data Types

From small data types, more complex data types can be created (recursively)

- **Arrays** → sequence of elements of same type  
  ```java
  int intArray[4];       // allocate array with 4 elements
  intArray[0] = 0x4711;   // set 1st element (index 0)
  ```

- **Pointers** → modifiable reference to a variable  
  ```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  
  ```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.