# System-Level Programming

## 7 Operations & Expressions

**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

# Arithmetic Operators [=Java]

- Can be used with all integer and floating-point types

| | | |
|---|---|---|
| + | addition | |
| − | subtraction | |
| ⋆ | multiplication | |
| / | division | |
| unary − | negative sign (e. g., $-a$) | ⇝ multiplication with −1 |
| unary + | positive sign (e. g., +3) | ⇝ no effect |

- Additionally only for integer types:

| | |
|---|---|
| % | modulo (remainder of division) |

# Increment/Decrement Operators

- Available for integer types and pointers

| | |
|---|---|
| ++ | increment (increase by 1) |
| −− | decrement (decrease by 1) |

- Left-side operator (prefix) ++x or --x
  - first, the value of variable x gets changed
  - then, the (new) value of x is used
- Right-side operator (postfix) x++ or x--
  - first, the (old) value of x is used
  - then, the value of x gets changed
- Examples

```
a = 10;
b = a++;  // b: 10, a: 11
c = ++a;  // c: 12, a: 12
```

# Comparison Operators [=Java]

- Comparison of two expressions

| | |
|---|---|
| `<` | less |
| `<=` | less or equal |
| `>` | greater |
| `>=` | greater or equal |
| `==` | identical (two equal signs!) |
| `! =` | unequal |

- Note: The result is of type **`int`** [≠Java]
  - Result: *false* ↦ 0
    *true* ↦ 1
  - The result can be used for calculations
- Examples

```
if (a >= 3) {···}
if (a == 3) {···}
return a * (a > 0);   // return 0 if a is negative
```

# Logic Operators [≈Java]

- Combining logical values (true / false), commutative

| Operator | Meaning | Example |
|---|---|---|
| && | "and" (conjunction) | *true* && *true* → *true*<br>*true* && *false* → *false*<br>*false* && *false* → *false* |
| \|\| | "or" (disjunction) | *true* \|\| *true* → *true*<br>*true* \|\| *false* → *true*<br>*false* \|\| *false* → *false* |
| ! | "not" (negation, unary) | ! *true* → *false*<br>! *false* → *true* |

- Note: operands and result are of type `int` [≠Java]
  - Operand (input parameter): $0 \mapsto$ *false*, $\neq 0 \mapsto$ *true*
  - Result: *false* $\mapsto 0$, *true* $\mapsto 1$

# Logical Operators – Evaluation [=Java]

- The evaluation of a logical expression is terminated as soon as the result is known
  - Let `int a = 5; int b = 3; int c = 7;`

$$\underbrace{\underbrace{a > b}_{1} \;||\; \underbrace{a > c}_{?}}_{1}$$

⟵ will not be evaluated since the first term already is *true*

$$\underbrace{\underbrace{a > c}_{0} \;\&\&\; \underbrace{a > b}_{?}}_{0}$$

⟵ will not be evaluated since the first term already is *false*

- This *short-circuit evaluation* can have surprising results if subexpressions have side effects!

```
int a = 5; int b = 3; int c = 7;
if ( a > c && !func(b) ) {···}      // func() will not be called
```

07-Operatoren_en

# Assignment Operators [=Java]

- General assignment operator (=)
  - assigns a value to a variable
  - example: `a = b + 23`
- Arithmetic assignment operators (+=, −=, ...)
  - shortened notation for modifying the value of a variable
  - example: `a += 23` is equivalent to `a = a + 23`
  - generally: `a` *op*`=` `b` is equivalent to `a = a` *op* `b`

    for *op* $\in \{ +, -, \star, /, \%, <<, >>, \&, \hat{}, | \}$
- Examples

```
int a = 8;
a += 8;     // a: 16
a %= 3;     // a: 1
```

# Assignments are Expressions!

- Assignments can be nested in more complex expressions
  - The result of an assignment is the assigned value.

```
int a, b, c;
a = b = c = 1;  // c: 1, b: 1, a: 1
```

  - The use of assignments in arbitrary expressions leads to **side effects**, which are not always obvious.

```
a += b += c;    // Value of a and b?
```

# Assignments are Expressions!

- Assignments can be nested in more complex expressions
  - The result of an assignment is the assigned value.

```
int a, b, c;
a = b = c = 1;  // c: 1, b: 1, a: 1
```

  - The use of assignments in arbitrary expressions leads to **side effects**, which are not always obvious.

```
a += b += c;    // Value of a and b?
```

**Particularly dangerous:** use of = instead of ==

In C, logical values are integers: $0 \mapsto$ *false*, $\emptyset \mapsto$ *true*

- Typical "rookie mistake" of control structures:

  `if (a = 6) {···} else {···} // BUG: if-branch is always taken!!!`

- Compiler possibly gives **no warning** about the construct as it is a valid expression! $\leadsto$ Programming bug is quite easy to miss!

# Bit Operations [=Java]

- Bit-wise operations of integers, commutative

| | | |
|---|---|---|
| & | bit-wise "and" (bit intersection) | 1 & 1 → 1<br>1 & 0 → 0<br>0 & 0 → 0 |
| \| | bit-wise "or" (bit unification) | 1 \| 1 → 1<br>1 \| 0 → 1<br>0 \| 0 → 0 |
| ^ | bit-wise "exclusive or" (bit antivalence) | 1 ^ 1 → 0<br>1 ^ 0 → 1<br>0 ^ 0 → 0 |
| ~ | bit-wise inversion (one's complement, unary) | ~ 1 → 0<br>~ 0 → 1 |

# Bit Operations (continued) [=Java]

- Shift operators on integers, not commutative

| | |
|---|---|
| `<<` | bit-wise left shift (on the right side, 0 bits are “inserted”) |
| `>>` | bit-wise right shift (on the left side, 0 bits are “inserted”) |

- Examples (let `x` be of type `uint8_t`)

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| `x=156` | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | `0x9c` |

- Shift operators on integers, not commutative

| | |
|---|---|
| `<<` | bit-wise left shift (on the right side, 0 bits are "inserted") |
| `>>` | bit-wise right shift (on the left side, 0 bits are "inserted") |

- Examples (let `x` be of type `uint8_t`)

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| `x=156` | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | `0x9c` |
| `~x` | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | `0x63` |

# Bit Operations (continued)

- Shift operators on integers, not commutative

| | |
|---|---|
| `<<` | bit-wise left shift (on the right side, 0 bits are “inserted”) |
| `>>` | bit-wise right shift (on the left side, 0 bits are “inserted”) |

- Examples (let `x` be of type `uint8_t`)

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| `x=156` | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | `0x9c` |
| `~x` | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | `0x63` |
| `7` | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | `0x07` |
| `x \| 7` | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | `0x9f` |

# Bit Operations (continued)

- Shift operators on integers, not commutative

| | |
|---|---|
| << | bit-wise left shift (on the right side, 0 bits are "inserted") |
| >> | bit-wise right shift (on the left side, 0 bits are "inserted") |

- Examples (let x be of type `uint8_t`)

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| x=156 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0x9c |
| ~x | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0x63 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0x07 |
| x \| 7 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0x9f |
| x & 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x04 |

# Bit Operations (continued) [=Java]

- Shift operators on integers, not commutative

  | | |
  |---|---|
  | << | bit-wise left shift (on the right side, 0 bits are "inserted") |
  | >> | bit-wise right shift (on the left side, 0 bits are "inserted") |

- Examples (let x be of type `uint8_t`)

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| x=156 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0x9c |
| ~x | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0x63 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0x07 |
| x \| 7 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0x9f |
| x & 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x04 |
| x ^ 7 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0x9B |

# Bit Operations (continued) [=Java]

- Shift operators on integers, not commutative

| | |
|---|---|
| << | bit-wise left shift (on the right side, 0 bits are “inserted”) |
| >> | bit-wise right shift (on the left side, 0 bits are “inserted”) |

- Examples (let `x` be of type `uint8_t`)

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| `x=156` | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | `0x9c` |
| `~x` | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | `0x63` |
| `7` | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | `0x07` |
| `x \| 7` | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | `0x9f` |
| `x & 7` | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | `0x04` |
| `x ^ 7` | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | `0x9B` |
| `x << 2` | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | `0x70` |

# Bit Operations (continued) [=Java]

- Shift operators on integers, not commutative

  `<<` bit-wise left shift (on the right side, 0 bits are "inserted")

  `>>` bit-wise right shift (on the left side, 0 bits are "inserted")

- Examples (let `x` be of type `uint8_t`)

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| `x=156` | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | `0x9c` |
| `~x` | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | `0x63` |
| `7` | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | `0x07` |
| `x \| 7` | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | `0x9f` |
| `x & 7` | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | `0x04` |
| `x ^ 7` | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | `0x9B` |
| `x << 2` | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | `0x70` |
| `x >> 1` | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | `0x4e` |

# Bit Operations – Usage

- By combining these operations, single bits are set/unset.

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| `PORTD` | ? | ? | ? | ? | ? | ? | ? | ? |

Bit 7 shall be changed without altering other bits!

# Bit Operations – Usage

- By combining these operations, single bits are set/unset.

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| `PORTD` | ? | ? | ? | ? | ? | ? | ? | ? | Bit 7 shall be changed without altering other bits! |
| `0x80` | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | One bit gets set by **or-operation** with a mask that only contains a 1 bit at the desired position |
| `PORTD \|= 0x80` | 1 | ? | ? | ? | ? | ? | ? | ? | |

07-Operatoren_en

# Bit Operations – Usage

- By combining these operations, single bits are set/unset.

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| `PORTD` | ? | ? | ? | ? | ? | ? | ? | ? | Bit 7 shall be changed without altering other bits! |
| `0x80` | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | One bit gets set by **or-operation** with a mask that only contains a 1 bit at the desired position |
| `PORTD \|= 0x80` | 1 | ? | ? | ? | ? | ? | ? | ? | |
| `~0x80` | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | One bit gets unset (set to 0) by **and-operation** with a mask that only contains a 0 bit at the desired position. |
| `PORTD &= ~0x80` | 0 | ? | ? | ? | ? | ? | ? | ? | |

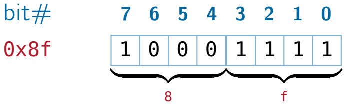
# Bit Operations – Usage

- By combining these operations, single bits are set/unset.

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| `PORTD` | ? | ? | ? | ? | ? | ? | ? | ? | Bit 7 shall be changed without altering other bits! |
| `0x80` | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | One bit gets set by **or-operation** with a mask that only contains a 1 bit at the desired position |
| `PORTD \|= 0x80` | 1 | ? | ? | ? | ? | ? | ? | ? | |
| `~0x80` | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | One bit gets unset (set to 0) by **and-operation** with a mask that only contains a 0 bit at the desired position. |
| `PORTD &= ~0x80` | 0 | ? | ? | ? | ? | ? | ? | ? | |
| `0x08` | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | Inversion of one bit by **xor-operation** with a mask that only contains a 1 bit at the desired position. |
| `PORTD ^= 0x08` | ? | ? | ? | ? | ¿ | ? | ? | ? | |

# Bit Operations – Usage (continued)

- Bit masks are usually given as hexadecimal literals.




hex digit represents half byte: *nibble*

# Bit Operations – Usage (continued)

- Bit masks are usually given as hexadecimal literals.

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x8f | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | 8 | | | | f | | | |

hex digit represents half byte: *nibble*

- For "thinkers in decimals", the left-shift notation is more suitable

```
PORTD |= (1<<7);    // set bit 7:  1<<7 --> 10000000
PORTD &= ~(1<<7);   // mask bit 7: ~(1<<7) --> 01111111
```

# Bit Operations – Usage (continued)

- Bit masks are usually given as hexadecimal literals.

| bit# | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x8f | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | 8 | | | | f | | | |

hex digit represents half byte: *nibble*

- For "thinkers in decimals", the left-shift notation is more suitable

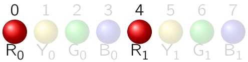
```
PORTD |= (1<<7);    // set bit 7:  1<<7 --> 10000000
PORTD &= ~(1<<7);   // mask bit 7: ~(1<<7) --> 01111111
```

- Combined with the or-operation, shifting ones works for complex masks

```
#include <led.h>
void main(void) {
  uint8_t mask = (1<<RED0) | (1<<RED1);

  sb_led_setMask (mask);

  while(1) ;
}
```

⇝ 

# Conditional Evaluation [≈Java]

- Formulation of conditions in expressions

  *expression*$_1$ **?** *expression*$_2$ **:** *expression*$_3$

  - first, *expression*$_1$ gets evaluated
    - *expression*$_1$ $\neq 0$ (*true*) $\leadsto$ *expression*$_2$ is the result
    - *expression*$_1$ $= 0$ (*false*) $\leadsto$ *expression*$_3$ is the result
  - `?:` is the only ternary (three-part) operator in C

- Example

```
int abs(int a) {
  // if (a<0) return -a; else return a;
  return (a<0) ? -a : a;
}
```

# Sequence Operator [≠Java]

- Sequencing of expressions

  *expression*$_1$ **,** *expression*$_2$

  - first, *expression*$_1$ gets evaluated
    $\leadsto$ side effects of *expression*$_1$ are visible for *expression*$_2$
  - the value of *expression*$_2$ is the result

- Use of the comma operator is often not required!
  (C-preprocessor macros with side effects)

# Associativity Rules of Operators [≈Java]

| | class | operators | associativity |
|---|---|---|---|
| 1 | function call, array access | `x() x[]` | `left` → `right` |
| | structure access | `x.y x->y` | |
| | post-increment/-decrement | `x++ x--` | |
| 2 | pre-increment/-decrement | `++x --x` | `right` → `left` |
| | unary operators | `+x -x ~x !x` | |
| | address, pointer | `& *` | |
| | type conversion (cast) | (<Typ>)x | |
| | type size | `sizeof(x)` | |
| 3 | multiplication, division, modulo | `* / %` | `left` → `right` |
| 4 | addition, subtraction | `+ -` | `left` → `right` |
| 5 | bit-wise shifts | `>> <<` | `left` → `right` |
| 6 | relational operators | `< <= > >=` | `left` → `right` |
| 7 | equality operators | `== !=` | `left` → `right` |
| 8 | bit-wise AND | `&` | `left` → `right` |
| 9 | bit-wise OR | \| | `left` → `right` |
| 10 | bit-wise XOR | `^` | `left` → `right` |
| 11 | conjunction | `&&` | `left` → `right` |
| 12 | disjunction | \|\| | `left` → `right` |
| 13 | conditional evaluation | `?:=` | `right` → `left` |
| 14 | assignment | `=` *op*`=` | `right` → `left` |
| 15 | sequence | `,` | `left` → `right` |

# Type Promotion in Expressions

- Operations are calculated *at least* with `int`-width
  - `short`- and `signed char`-operands are “promoted” implicitly (↪ *Integer Promotion*)
  - Only the result will then be promoted/cut off to match the target type

```
int8_t a=100, b=3, c=4, res;   // range: -128 --> +127

res    =  a    *  b    /  c  ; // promotion to int: 300 fits in!
int8_t: 75   int: 100   int: 3   int: 4
                int: 300
                     int: 75
```

# Type Promotion in Expressions (continued)

- In general, the *largest* involved width is used

```
int8_t a=100, b=3, res;  // range: -128 --> +127
int32_t c=4;             // range: -2147483648 --> +2147483647

res = a * b / c; // promotion to int32_t
```

- res → int8_t: 75
- a → int: 100
- b → int: 3
- a * b → int: 300
- → int32_t: 300
- a * b / c → int32_t: 75

# Type Casting in Expressions (continued)

- Floating-point types are thereby considered to be “larger” than integer types
- All floating point operations are *at least* calculated with `double` width

```
int8_t a=100, b=3, res;          // range: -128 --> +127

res  =  a  *  b  /  4.0f  ;  // promotion to double
```

- res: int8_t: 75
- a: int: 100
- b: int: 3
- 4.0f: double 4.0
- a * b: int: 300
- then: double: 300.0
- a * b / 4.0f: double: 75.0

# Type Promotion in Expressions (continued)

- `unsigned` types are also considered "larger" than `signed` types

```
int s = -1, res;          // range: -32768 --> +32767
unsigned u = 1;           // range: 0 --> 65535

res  =   s    < u;        // promotion to unsigned: -1 --> 65535
```



⇝ Surprising results when using negative values!

⇝ Avoid mixing `signed` and `unsigned` operands!

# Type Casting in Expressions – Type Casts

- By using the type cast operator, an expression is converted into a target type.
- Casting is explicit type promotion.

**(***type***)** *expression*

```
int s = -1, res;        // range: -32768 --> +32767
unsigned u = 1;         // range: 0 --> 65535

res  = s < (int) u;     // cast u to int
int: 1        int: 1
          int: 1
```

# References

[GDI] Frank Bauer. *Grundlagen der Informatik*. Vorlesung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 5, 2015 (jährlich). URL: `https://gdi.cs.fau.de/w15/material`.