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

- Can be used with all integer and floating-point types
  - + : addition
  - - : subtraction
  - * : multiplication
  - / : division
  - unary - : negative sign (e.g., \(-a\)) \(\sim\) multiplication with \(-1\)
  - unary + : positive sign (e.g., \(+3\)) \(\sim\) 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

Comparison of two expressions

<  
   less

<=
   less or equal

>
   greater

>=
   greater or equal

==
   identical (two equal signs!)

!=
   unequal

Note: The result is of type \texttt{int}

- Result:
  
  \texttt{false} \mapsto 0
  \texttt{true} \mapsto 1

- The result can be used for calculations

Examples

\begin{verbatim}
if (a >= 3) {···}
if (a == 3) {···}
return a * (a > 0); // return 0 if a is negative
\end{verbatim}
### Logic Operators

**Combining logical values (true / false), commutative**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
</table>
| `&&`     | "and" (conjunction) | `true && true` → `true`  
|          |               | `true && false` → `false`  
|          |               | `false && false` → `false`  |
| `||`     | "or" (disjunction) | `true || true` → `true`  
|          |               | `true || false` → `true`  
|          |               | `false || false` → `false` |
| `!`      | "not" (negation, unary) | `! true` → `false`  
|          |               | `! false` → `true`  |

**Note:** operands and result are of type `int`

- **Operand** (input parameter):
  - `0` ↦ `false`
  - `≠0` ↦ `true`

- **Result:**
  - `false` ↦ `0`
  - `true` ↦ `1`
The evaluation of a logical expression is terminated as soon as the result is known.

Let \( \text{int } a = 5; \text{ int } b = 3; \text{ int } c = 7; \)

\[
\begin{align*}
\text{if } a > b \quad &\text{||} \quad a > c \quad \leftarrow \quad \text{will not be evaluated since the first term already is true} \\
&\downarrow \downarrow \downarrow \downarrow \\
&1 \quad ? \quad 1
\end{align*}
\]

\[
\begin{align*}
\text{if } a > c \quad &\& \quad a > b \quad \leftarrow \quad \text{will not be evaluated since the first term already is false} \\
&\downarrow \downarrow \downarrow \downarrow \\
&0 \quad ? \quad 0
\end{align*}
\]

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

\[
\begin{align*}
\text{int } a = 5; \text{ int } b = 3; \text{ int } c = 7; \\
\text{if } ( a > c \quad &\& \quad !\text{func}(b) ) \{\cdots\} \quad \text{\texttt{// func() will not be called}
\end{align*}
\]
Assignment Operators

- 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 ∈ { +, −, *, /, %, <<, >>, &, ^, | }`

Examples

```java
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.

```c
int a, b, c;
int 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.

```c
int a, b, c;
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.

```c
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.

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

Particularly dangerous: use of = instead of ==

In C, logical values are integers: 0 \rightarrow false, \emptyset \rightarrow true

- Typical “rookie mistake” of control structures:
  ```c
  if (a = 6) {⋯} else {⋯} // BUG: if-branch is always taken!!!
  ```

- Compiler possibly gives no warning about the construct as it is a valid expression! Programming bug is quite easy to miss!
## Bit Operations

### Bit-wise operations of integers, commutative

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>bit-wise “and” (bit intersection)</td>
<td>$1 &amp; 1 \rightarrow 1$, $1 &amp; 0 \rightarrow 0$, $0 &amp; 0 \rightarrow 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bit-wise “or” (bit unification)</td>
<td>$1 \mid 1 \rightarrow 1$, $1 \mid 0 \rightarrow 1$, $0 \mid 0 \rightarrow 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise “exclusive or” (bit antivalence)</td>
<td>$1 \wedge 1 \rightarrow 0$, $1 \wedge 0 \rightarrow 1$, $0 \wedge 0 \rightarrow 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>bit-wise inversion (one’s complement, unary)</td>
<td>$\sim 1 \rightarrow 0$, $\sim 0 \rightarrow 1$</td>
</tr>
</tbody>
</table>
### 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`)

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=156</code></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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`)

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=156</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>~x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

0x9c
0x63
Bit Operations (continued)

Shift operators on integers, not commutative

<p>&lt;&lt; bit-wise left shift (on the right side, 0 bits are “inserted”)
&gt;&gt; bit-wise right shift (on the left side, 0 bits are “inserted”)</p>

Examples (let x be of type uint8_t)

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=156</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>~x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

© kls
System-Level Programming (ST 24)
7 Operations & Expressions – Bit Operations
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`)

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=156</code></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>~x</code></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>`x</td>
<td>7`</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>x &amp; 7</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>0x9c</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>0x63</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>0x07</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>0x9f</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>0x04</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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`)

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=156</code></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>~<code>x</code></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>7</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>`x</td>
<td>7`</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>x &amp; 7</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>x ^ 7</code></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>


© klsystems
System-Level Programming (ST 24)  7 Operations & Expressions – Bit Operations
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`)

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=156</code></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>~x</code></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><code>7</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>`x</td>
<td>7`</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>x &amp; 7</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>x ^ 7</code></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>x &lt;&lt; 2</code></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
| 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 | 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.

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Bit 7 shall be changed without altering other bits!
Bit Operations – Usage

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

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Bit 7 shall be changed without altering other bits!

PORTD |= 0x80

| 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

| 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

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

<table>
<thead>
<tr>
<th>bit#</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80</td>
<td>1 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Bit 7 shall be changed without altering other bits!

One bit gets set by or-operation with a mask that only contains a 1 bit at the desired position.

PORTD |= 0x80

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
Bit Operations – Usage

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

<table>
<thead>
<tr>
<th>bit#</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
</table>

Bit 7 shall be changed without altering other bits!

<table>
<thead>
<tr>
<th>0x80</th>
<th>1 0 0 0 0 0 0 0</th>
</tr>
</thead>
</table>

One bit gets set by or-operation with a mask that only contains a 1 bit at the desired position.

<table>
<thead>
<tr>
<th>~0x80</th>
<th>0 1 1 1 1 1 1 1</th>
</tr>
</thead>
</table>

One bit gets unset (set to 0) by and-operation with a mask that only contains a 0 bit at the desired position.

<table>
<thead>
<tr>
<th>0x08</th>
<th>0 0 0 0 1 0 0 0</th>
</tr>
</thead>
</table>

Inversion of one bit by xor-operation with a mask that only contains a 1 bit at the desired position.
Bit masks are usually given as hexadecimal literals.

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8f</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

hex digit represents half byte: *nibble*
Bit masks are usually given as hexadecimal literals.

<table>
<thead>
<tr>
<th>bit#</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8f</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

hex digit represents half byte: *nibble*

For “thinkers in decimals”, the left-shift notation is more suitable:

```c
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#

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

0x8f

\[\begin{array}{c|c|c|c|c|c|c|c}
8 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}\]

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

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

© klsw  System-Level Programming (ST 24)  7 Operations & Expressions  – Bit Operations 7–19
Conditional Evaluation

Formulation of conditions in expressions

\( \text{expression}_1 \ ? \ \text{expression}_2 \ : \ \text{expression}_3 \)

- first, \( \text{expression}_1 \) gets evaluated
  - \( \text{expression}_1 \neq 0 \) (true) \( \Rightarrow \) \( \text{expression}_2 \) is the result
  - \( \text{expression}_1 = 0 \) (false) \( \Rightarrow \) \( \text{expression}_3 \) is the result

?: is the only ternary (three-part) operator in C

Example

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

- Sequencing of expressions
  \[ \text{expression}_1 \ , \ \text{expression}_2 \]
  - first, \( \text{expression}_1 \) gets evaluated
    \[ \sim \text{side effects of } \text{expression}_1 \text{ are visible for } \text{expression}_2 \]
  - the value of \( \text{expression}_2 \) is the result

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function call, array access</td>
<td><code>x()</code> <code>x[]</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Structure access</td>
<td><code>x.y</code> <code>x-&gt;y</code></td>
<td></td>
</tr>
<tr>
<td>Post-increment/-decrement</td>
<td><code>x++</code> <code>x--</code></td>
<td></td>
</tr>
<tr>
<td>Pre-increment/-decrement</td>
<td><code>++x</code> <code>-x</code></td>
<td>right → left</td>
</tr>
<tr>
<td>Unary operators</td>
<td><code>+x</code> <code>-x</code> <code>~x</code> <code>!x</code> <code>&amp;</code> <code>*</code></td>
<td></td>
</tr>
<tr>
<td>Address, pointer</td>
<td><code>(Typ)x</code> <code>sizeof(x)</code></td>
<td></td>
</tr>
<tr>
<td>Type conversion (cast)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplication, division, modulo</td>
<td><code>*</code> <code>/</code> <code>%</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Addition, subtraction</td>
<td><code>+</code> <code>-</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Bit-wise shifts</td>
<td><code>&gt;&gt;</code> <code>&lt;&lt;</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Relational operators</td>
<td><code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Equality operators</td>
<td><code>==</code> <code>!=</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Bit-wise AND</td>
<td><code>&amp;</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Bit-wise OR</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>Bit-wise XOR</td>
<td><code>^</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Conjunction</td>
<td><code>&amp;&amp;</code></td>
<td>left → right</td>
</tr>
<tr>
<td>Disjunction</td>
<td>`</td>
<td></td>
</tr>
<tr>
<td>Conditional evaluation</td>
<td><code>?:=</code></td>
<td>right → left</td>
</tr>
<tr>
<td>Assignment</td>
<td><code>op=</code></td>
<td>right → left</td>
</tr>
<tr>
<td>Sequence</td>
<td><code>=</code></td>
<td>left → right</td>
</tr>
</tbody>
</table>
Type Promotion in Expressions

Operations are calculated *at least* with **int**-width

- **short**- and **signed char**-operands are “promoted” implicitly
  
  \[ \rightarrow \text{Integer Promotion} \]

- Only the result will then be promoted/cut off to match the target type

```c
int8_t a=100, b=3, c=4, res; // range: -128 --> +127
res = a * b / c ; // promotion to int: 300 fits in!
```

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>75</td>
</tr>
<tr>
<td>int</td>
<td>100</td>
</tr>
<tr>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>300</td>
</tr>
<tr>
<td>int</td>
<td>75</td>
</tr>
</tbody>
</table>
```
In general, the *largest* involved width is used

```c
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
```

```
int8_t: 75
  int: 100
    int: 3
      int: 300
int32_t: 300
  int32_t: 75
```
Floating-point types are thereby considered to be “larger” than integer types.

All floating point operations are *at least* calculated with double width.

```c
int8_t a = 100, b = 3, res; // range: -128 --> +127
res = a * b / 4.0f; // promotion to double
```

- `int8_t: 75`
- `int: 100`
- `int: 3`
- `double: 4.0`
- `int: 300`
- `double: 300.0`
- `double: 75.0`
unsigned types are also considered “larger” than signed types

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

```plaintext
int: 0  unsigned: 65535
```

```plaintext
unsigned: 0
```

〜 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.

\[(\text{type}) \ expression\]

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

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

int: 1
```
References