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

\[ \begin{align*}
++ & \quad \text{increment (increase by 1)} \\
-- & \quad \text{decrement (decrease by 1)}
\end{align*} \]

Left-side operator (prefix)

\[ \begin{align*}
++x \quad \text{or} \quad --x
\end{align*} \]

- first, the value of variable \( x \) gets changed
- then, the (new) value of \( x \) is used

Right-side operator (postfix)

\[ \begin{align*}
x++ \quad \text{or} \quad x--
\end{align*} \]

- first, the (old) value of \( x \) is used
- then, the value of \( x \) gets changed

Examples

\[ \begin{align*}
a & = 10; \\
b & = a++; \quad // \ b: 10, \ a: 11 \\
c & = ++a; \quad // \ c: 12, \ a: 12
\end{align*} \]
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 int

- Result:  
  - `false`  $\mapsto$  0
  - `true`  $\mapsto$  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

Combining logical values (true / false), commutative

& & "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
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;`  

\[
\begin{align*}
\text{a} & > \text{b} \quad \text{||} \quad \text{a} & > \text{c} \\
1 & \quad \quad \quad ? \quad \quad \quad 1 \\
\end{align*}
\]

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

\[
\begin{align*}
\text{a} & > \text{c} \quad \text{&&} \quad \text{a} & > \text{b} \\
0 & \quad \quad \quad ? \quad \quad \quad 0 \\
\end{align*}
\]

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

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

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

General assignment operator (=)
- assigns a value to a variable
- example:  \( a = b + 23 \)

Arithmetic assignment operators (\( +=, -=, \ldots \))
- 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 \( op \in \{+,-,\times,\div,\%,<,>,<<,>>,\&,\^,|\} \)

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;
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?
```
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 \(\mapsto\) false, \(\not=\) 0 \(\mapsto\) 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!  \(\mapsto\) Programming bug is quite easy to miss!
### Bit-wise operations of integers, commutative

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
</table>
| `&`      | bit-wise “and”               | $1 \& 1 \rightarrow 1$  
|          | (bit intersection)           | $1 \& 0 \rightarrow 0$  
|          |                               | $0 \& 0 \rightarrow 0$  |
| `|`       | bit-wise “or”                | $1 \mid 1 \rightarrow 1$  
|          | (bit unification)            | $1 \mid 0 \rightarrow 1$  
|          |                               | $0 \mid 0 \rightarrow 0$  |
| `^`      | bit-wise “exclusive or”      | $1 \wedge 1 \rightarrow 0$  
|          | (bit antivalence)            | $1 \wedge 0 \rightarrow 1$  
|          |                               | $0 \wedge 0 \rightarrow 0$  |
| `~`      | bit-wise inversion           | $\sim 1 \rightarrow 0$  
|          | (one’s complement, unary)    | $\sim 0 \rightarrow 1$  |
Shift operators on integers, not commutative

\[<\langle \text{bit-wise left shift (on the right side, 0 bits are “inserted”) }\]
\[\gg \text{ bit-wise right shift (on the left side, 0 bits are “inserted”) }\]

Examples (let \(x\) be of type \texttt{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>
</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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x9c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x63</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>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>
<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>

0x9c, 0x63, 0x07, 0x9f
Shift operators on integers, not commutative

\[\ll\] bit-wise left shift (on the right side, 0 bits are “inserted”)
\[\gg\] bit-wise right shift (on the left side, 0 bits are “inserted”)

Examples (let \(x\) be of type \(\text{uint8}_t\))

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

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

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

| PORTD |= 0x80 | 1 | ? | ? | ? | ? | ? | ? |

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**

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.
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!

<table>
<thead>
<tr>
<th>$\text{0x80}$</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>
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
```

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) ;
}
```
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.

<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
```

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) ;
}
```
Conditional Evaluation

Formulation of conditions in expressions

\[ expression_1 \ ? \ expression_2 : expression_3 \]

- first, \( expression_1 \) gets evaluated
  - \( expression_1 \neq 0 \) (true) \( \Rightarrow expression_2 \) is the result
  - \( expression_1 = 0 \) (false) \( \Rightarrow expression_3 \) 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
  ~ side effects of \( \text{expression}_1 \) 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>1 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 x-&gt;y</code></td>
<td></td>
</tr>
<tr>
<td>post-increment/-decrement</td>
<td><code>x++ x--</code></td>
<td></td>
</tr>
<tr>
<td>2 pre-increment/-decrement</td>
<td><code>++x -x</code></td>
<td>right → left</td>
</tr>
<tr>
<td>unary operators</td>
<td><code>+x -x ~x !x</code></td>
<td></td>
</tr>
<tr>
<td>address, pointer</td>
<td><code>&amp; *</code></td>
<td></td>
</tr>
<tr>
<td>type conversion (cast)</td>
<td><code>(&lt;Typ&gt;)x</code></td>
<td></td>
</tr>
<tr>
<td>type size</td>
<td><code>sizeof(x)</code></td>
<td></td>
</tr>
<tr>
<td>3 multiplication, division, modulo</td>
<td><code>* / %</code></td>
<td>left → right</td>
</tr>
<tr>
<td>4 addition, subtraction</td>
<td><code>+ -</code></td>
<td>left → right</td>
</tr>
<tr>
<td>5 bit-wise shifts</td>
<td><code>&gt;&gt; &lt;&lt;</code></td>
<td>left → right</td>
</tr>
<tr>
<td>6 relational operators</td>
<td><code>&lt; &lt;= &gt; &gt;=</code></td>
<td>left → right</td>
</tr>
<tr>
<td>7 equality operators</td>
<td><code>== !=</code></td>
<td>left → right</td>
</tr>
<tr>
<td>8 bit-wise AND</td>
<td><code>&amp;</code></td>
<td>left → right</td>
</tr>
<tr>
<td>9 bit-wise OR</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>10 bit-wise XOR</td>
<td><code>^</code></td>
<td>left → right</td>
</tr>
<tr>
<td>11 conjunction</td>
<td><code>&amp;&amp;</code></td>
<td>left → right</td>
</tr>
<tr>
<td>12 disjunction</td>
<td>`</td>
<td></td>
</tr>
<tr>
<td>13 conditional evaluation</td>
<td><code>?:=</code></td>
<td>right → left</td>
</tr>
<tr>
<td>14 assignment</td>
<td><code>= op=</code></td>
<td>right → left</td>
</tr>
<tr>
<td>15 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
    *(→ 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!
```

- `int8_t`: 75
  - `int`: 100
  - `int`: 3
  - `int`: 4
  - `int`: 300
  - `int`: 75
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
```

Surprising results when using negative values!
Avoid mixing signed and unsigned operands!
By using the type cast operator, an expression is converted into a target type.

Casting is explicit type promotion.

\((\text{type}) \; \text{expression}\)

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

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

```
int: 1

int: 1

int: 1
```
References