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

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

Comparison of two expressions

<   less
\leq   less or equal
>   greater
\geq   greater or equal
==   identical (two equal signs!)
!=   unequal

Note: The result is of type int

- Result: false \rightarrow 0
true \rightarrow 1

- The result can be used for calculations

Examples

```java
if (a >= 3) {...}
if (a == 3) {...}
return a * (a > 0);  // return 0 if a is negative
```
## Logic Operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Truth Table</th>
</tr>
</thead>
</table>
| `&&`    | “and”       | true && true → true  
               true && false → false  
               false && false → false |
| `||`    | “or”        | true || true → true  
               true || false → true  
               false || false → false |
| `!`     | “not”       | !true → false  
               !false → true |

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

- **Operands (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 \( \text{int } a = 5; \text{ int } b = 3; \text{ int } c = 7; \)

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

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

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 (+=, −=, …)
- 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 \( op \in \{ +, -, *, /, \%, <, >, >>, \&, ^{\land}, | \} \)

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.
    
    ```
    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, /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!

Programming bug is quite easy to miss!
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 ↦ false, ∅ ↦ 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>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>bit-wise “and” (bit intersection)</td>
<td>1 &amp; 1 \rightarrow 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 &amp; 0 \rightarrow 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 &amp; 0 \rightarrow 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit-wise “or” (bit unification)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise “exclusive or” (bit antivalence)</td>
<td>1 ^ 1 \rightarrow 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 ^ 0 \rightarrow 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 ^ 0 \rightarrow 0</td>
</tr>
<tr>
<td>~</td>
<td>bit-wise inversion (one’s complement, unary)</td>
<td>~ 1 \rightarrow 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~ 0 \rightarrow 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>x=156</td>
<td>1 0 0 1 1 1 0 0</td>
</tr>
<tr>
<td></td>
<td>0x9c</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 \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>
<tr>
<td>( \sim 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>
</tbody>
</table>

\( \text{0x9c} \)

\( \text{0x63} \)
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>
<tr>
<td>0x9c</td>
<td>0x63</td>
<td>0x07</td>
<td>0x9f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>~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>
<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</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>
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>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>
<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>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>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>
</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>$\sim x$</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>
<tr>
<td>$x &amp; 7$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>$x &lt;&lt; 2$</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>$x &gt;&gt; 1$</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>
</tbody>
</table>
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>
<tbody>
<tr>
<td>PORTD</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</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 6 5 4 3 2 1 0</th>
</tr>
</thead>
</table>

Bit 7 shall be changed without altering other bits!

\[
\text{PORTD} \ OR \ 0x80
\]

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

\[
\text{PORTD} \ OR \ 0x80
\]

\[
\text{PORTD} \ SET \ 0x08
\]

Inversion of one bit by \textbf{xor-operation} with a mask that only contains a 1 bit at the desired position.

\[
\text{PORTD} \ XOR \ 0x08
\]
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 → 0x08

<table>
<thead>
<tr>
<th>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>
<tbody>
<tr>
<td>PORTD</td>
<td>= 0x80</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

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

~0x80 → 0x08

<table>
<thead>
<tr>
<th>~0x80</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>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.
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>PORTD</td>
<td>?? ?? ?? ?? ?? ??</td>
</tr>
</tbody>
</table>

- **PORTD |= 0x80**
  - 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 ^= 0x08**
  - Inversion of one bit by **xor-operation** with a mask that only contains a 1 bit at the desired position.

- Bit 7 shall be changed without altering other bits!
Bit masks are usually given as hexadecimal literals.

\[
\begin{array}{c|ccccccc}
\text{bit\#} & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
0x8f & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

hex digit represents half byte: \textit{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 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

\[ \text{expression}_1 \ ? \ \text{expression}_2 : \ \text{expression}_3 \]

- first, \( \text{expression}_1 \) gets evaluated
  - \( \text{expression}_1 \neq 0 \) (true) \( \leadsto \) \( \text{expression}_2 \) is the result
  - \( \text{expression}_1 = 0 \) (false) \( \leadsto \) \( \text{expression}_3 \) 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
  \[ expression_1, expression_2 \]
  - first, \( expression_1 \) gets evaluated
  - 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

<table>
<thead>
<tr>
<th>Class</th>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 function call, array access</td>
<td>x() x[]</td>
<td>left → right</td>
</tr>
<tr>
<td>structure access</td>
<td>x.y x-&gt;y</td>
<td></td>
</tr>
<tr>
<td>post-increment/-decrement</td>
<td>x++ x--</td>
<td></td>
</tr>
<tr>
<td>2 pre-increment/-decrement</td>
<td>++x -x</td>
<td>right → left</td>
</tr>
<tr>
<td>unary operators</td>
<td>+x -x ~x !x</td>
<td></td>
</tr>
<tr>
<td>address, pointer</td>
<td>&amp; *</td>
<td></td>
</tr>
<tr>
<td>type conversion (cast)</td>
<td>(&lt;Typ&gt;)x</td>
<td></td>
</tr>
<tr>
<td>type size</td>
<td>sizeof(x)</td>
<td></td>
</tr>
<tr>
<td>3 multiplication, division, modulo</td>
<td>* / %</td>
<td>left → right</td>
</tr>
<tr>
<td>4 addition, subtraction</td>
<td>+ -</td>
<td>left → right</td>
</tr>
<tr>
<td>5 bit-wise shifts</td>
<td>&gt;&gt; &lt;&lt;</td>
<td>left → right</td>
</tr>
<tr>
<td>6 relational operators</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left → right</td>
</tr>
<tr>
<td>7 equality operators</td>
<td>== !=</td>
<td>left → right</td>
</tr>
<tr>
<td>8 bit-wise AND</td>
<td>&amp;</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>^</td>
<td>left → right</td>
</tr>
<tr>
<td>11 conjunction</td>
<td>&amp;&amp;</td>
<td>left → right</td>
</tr>
<tr>
<td>12 disjunction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 conditional evaluation</td>
<td>?:=</td>
<td>right → left</td>
</tr>
<tr>
<td>14 assignment</td>
<td>= op=</td>
<td>right → left</td>
</tr>
<tr>
<td>15 sequence</td>
<td>,</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
```
Type Casting in Expressions  (continued)

- Floating-point types are thereby considered to be “larger” than integer types
- All floating point operations are \textit{at least} calculated with \texttt{double} width

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

- \texttt{int8_t}: 75
- \texttt{int}: 100
- \texttt{int}: 3
- \texttt{double}: 4.0
- \texttt{int}: 300
- \texttt{double}: 300.0
- \texttt{double}: 75.0
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
```

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

```
unsigned: 0
```

~ 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