2. Integers

Evaluation of Arithmetic Expressions, Associativity and Precedence, Arithmetic Operators, Domain of Types int, unsigned int

Example: power8.cpp

```cpp
int a; // Input
int r; // Result

std::cout << "Compute a^8 for a = ?";
std::cin >> a;

r = a * a; // r = a^2
r = r * r; // r = a^4

std::cout << "a^8 = " << r*r << '\n';
```

Terminology: L-Values and R-Values

L-Wert ("Left of the assignment operator")
- Expression identifying a memory location
- For example a variable
  (we’ll see other L-values later in the course)
- Value is the content at the memory location according to the type of the expression.
- L-Value can change its value (e.g. via assignment)

R-Wert ("Right of the assignment operator")
- Expression that is no L-value
- Example: integer literal 0
- Any L-Value can be used as R-Value (but not the other way round)
  ...
- ... by using the value of the L-value
  (e.g. the L-value a could have the value 2, which is then used as an R-value)
- An R-Value cannot change its value
L-Values and R-Values

std::cout << "Compute a^8 for a = ? ";
int a;
std::cin >> a;
int r = a * a; // r = a^2
r = r * r; // r = a^4
std::cout << a << '^' << 8 << " = " << r * r << ".
return 0;

R-Value (expression + address)
L-value (expression + address)
R-Value
R-Value (expression that is not an L-value)

Celsius to Fahrenheit

// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.
#include <iostream>

int main() {
    // Input
    std::cout << "Temperature in degrees Celsius = ? ";
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
               << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}

9 * celsius / 5 + 32

- Arithmetic expression,
- contains three literals, a variable, three operator symbols

How to put the expression in parentheses?

Rule 1: precedence

Multiplicative operators (*, /, %) have a higher precedence ("bind more strongly") than additive operators (+, -)
**Associativity**

**Rule 2: Associativity**

Arithmetic operators (*, /, %, +, -) are left associative: operators of same precedence evaluate from left to right.

\[
9 \times \text{celsius} / 5 + 32 \quad \Rightarrow \quad ((9 \times \text{celsius}) / 5) + 32
\]

**Arity**

**Rule 3: Arity**

Unary operators +, − first, then binary operators +, −.

\[-3 - 4 \quad \Rightarrow \quad (-3) - 4\]

**Parentheses**

Any expression can be put in parentheses by means of
- associativities
- precedences
- arities (number of operands)

of the operands in an unambiguous way (Details in the lecture notes).

**Expression Trees**

Parentheses yield the expression tree

\[
(((9 \times \text{celsius}) / 5) + 32)
\]
Evaluation Order

"From top to bottom" in the expression tree

Evaluation Order

Order is not determined uniquely:

Expression Trees – Notation

Common notation: root on top

Evaluation Order – more formally

- Valid order: any node is evaluated after its children
- In C++, the valid order to be used is not defined.
- "Good expression": any valid evaluation order leads to the same result.
- Example for a "bad expression": a*(a=2)
Guideline

Avoid modifying variables that are used in the same expression more than once.

### Evaluation order

### Arithmetic operations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary +</td>
<td>+</td>
<td>1</td>
<td>16 right</td>
</tr>
<tr>
<td>Negation</td>
<td>-</td>
<td>1</td>
<td>16 right</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>2</td>
<td>14 left</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>2</td>
<td>14 left</td>
</tr>
<tr>
<td>Modulo</td>
<td>%</td>
<td>2</td>
<td>14 links</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>2</td>
<td>13 left</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>2</td>
<td>13 left</td>
</tr>
</tbody>
</table>

All operators: [R-value ×] R-value → R-value

### Interlude: Assignment expression – in more detail

- Already known: `a = b` means Assignment of `b` (R-value) to `a` (L-value). Returns: L-value
- What does `a = b = c` mean?
- Answer: assignment is right-associative

```
a = b = c  ↔  a = (b = c)
```

Example multiple assignment:
```
a = b = 0  ⇒  b=0; a=0
```

### Division

- Operator `/` implements integer division

```
5 / 2 has value 2
```

In `fahrenheit.cpp`
```
9 * celsius / 5 + 32
```

15 degrees Celsius are 59 degrees Fahrenheit

- Mathematically equivalent… but not in C++!

```
9 / 5 * celsius + 32
```

15 degrees Celsius are 47 degrees Fahrenheit
Loss of Precision

Guideline
- Watch out for potential loss of precision
- Postpone operations with potential loss of precision to avoid “error escalation”

Division and Modulo
- Modulo-operator computes the rest of the integer division
  
  \[ \frac{5}{2} \text{ has value 2, } \quad 5 \mod{2} \text{ has value 1.} \]

- It holds that:
  
  \[ \left( \frac{a}{b} \right) \times b + a \mod{b} \text{ has the value of } a. \]

- From the above one can conclude the results of division and modulo with negative numbers

Increment and decrement
- Increment / Decrement a number by one is a frequent operation
- works like this for an L-value:

\[
expr = expr + 1.
\]

Disadvantages
- relatively long
- \( expr \) is evaluated twice
  - Later: L-valued expressions whose evaluation is “expensive”
  - \( expr \) could have an effect (but should not, cf. guideline)

In-/Decrement Operators
- Post-Increment
  \[
  expr++
  \]
  Value of \( expr \) is increased by one, the old value of \( expr \) is returned (as R-value)

- Pre-increment
  \[
  ++expr
  \]
  Value of \( expr \) is increased by one, the new value of \( expr \) is returned (as L-value)

- Post-Dekrement
  \[
  expr--
  \]
  Value of \( expr \) is decreased by one, the old value of \( expr \) is returned (as R-value)

- Prä-Dekrement
  \[
  --expr
  \]
  Value of \( expr \) is decreased by one, the new value of \( expr \) is returned (as L-value)
### In-/decrement Operators

<table>
<thead>
<tr>
<th>use</th>
<th>arity</th>
<th>prec</th>
<th>assoz</th>
<th>L-R-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post-increment</td>
<td>expr++</td>
<td>1</td>
<td>17</td>
<td>left L-value → R-value</td>
</tr>
<tr>
<td>Pre-increment</td>
<td>++expr</td>
<td>1</td>
<td>16</td>
<td>right L-value → L-value</td>
</tr>
<tr>
<td>Post-decrement</td>
<td>expr--</td>
<td>1</td>
<td>17</td>
<td>left L-value → R-value</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>--expr</td>
<td>1</td>
<td>16</td>
<td>right L-value → L-value</td>
</tr>
</tbody>
</table>

#### Example

```cpp
int a = 7;
std::cout << ++a << "\n"; // 8
std::cout << a++ << "\n"; // 8
std::cout << a << "\n"; // 9
```

### In-/Decrement Operators

Is the expression `++expr;` ← we favour this equivalent to `expr++;`?

Yes, but

- Pre-increment can be more efficient (old value does not need to be saved)
- Post In-/Decrement are the only left-associative unary operators (not very intuitive)

#### Arithmetic Assignments

\[
\text{a += b} \quad \leftrightarrow \quad \text{a = a + b}
\]

analogously for -, *, / and %
### Arithmetic Assignments

<table>
<thead>
<tr>
<th>Gebrauch</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>+= expr1 += expr2</td>
<td>expr1 = expr1 + expr2</td>
</tr>
<tr>
<td>-= expr1 -= expr2</td>
<td>expr1 = expr1 - expr2</td>
</tr>
<tr>
<td>*= expr1 *= expr2</td>
<td>expr1 = expr1 * expr2</td>
</tr>
<tr>
<td>/= expr1 /= expr2</td>
<td>expr1 = expr1 / expr2</td>
</tr>
<tr>
<td>%= expr1 %= expr2</td>
<td>expr1 = expr1 % expr2</td>
</tr>
</tbody>
</table>

Arithmetic expressions evaluate expr1 only once. Assignments have precedence 4 and are right-associative.

### Binary Number Representations

Binary representation (Bits from \{0, 1\})

\[b_nb_{n-1}\ldots b_1b_0\]

corresponds to the number

\[b_n \cdot 2^n + \ldots + b_1 \cdot 2 + b_0\]

**Example:** 101011 corresponds to 43.

**Least Significant Bit (LSB)**

**Most Significant Bit (MSB)**

### Computing Tricks

- Estimate the orders of magnitude of powers of two:\(^2\):

  \[2^{10} = 1024 = 1\text{Ki} \approx 10^3,\]
  \[2^{20} = 1\text{Mi} \approx 10^6,\]
  \[2^{30} = 1\text{Gi} \approx 10^9,\]
  \[2^{32} = 4 \cdot (1024)^3 = 4\text{Gi},\]
  \[2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.\]

### Hexadecimal Numbers

Numbers with base 16

\[h_nh_{n-1}\ldots h_1h_0\]

corresponds to the number

\[h_n \cdot 16^n + \ldots + h_1 \cdot 16 + h_0.\]

**notation in C++:** prefix 0x

**Example:** 0xff corresponds to 255.

---

\(^2\)Decimal vs. binary units: MB - Megabyte vs. MiB - Megabyte (etc.)

kilo (K, Ki) – mega (M, Mi) – giga (G, Gi) – tera (T, Ti) – peta (P, Pi) – exa (E, Ei)

---

<table>
<thead>
<tr>
<th>Hex Nibbles</th>
<th>hex</th>
<th>bin</th>
<th>dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1010</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1011</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1100</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1101</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1110</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1111</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
Why Hexadecimal Numbers?

- A Hex-Nibble requires exactly 4 bits. Numbers 1, 2, 4, and 8 represent bits 0, 1, 2, and 3.
- "compact representation of binary numbers"

Example: Hex-Colors

```
#00FF00
```

r  g  b

Domain of Type `int`

```
// Output the smallest and the largest value of type int.
#include <iostream>
#include <limits>

int main() {
    std::cout << "Minimum int value is "
    << std::numeric_limits<int>::min() << "\n"
    << "Maximum int value is "
    << std::numeric_limits<int>::max() << "\n";
    return 0;
}
```

Minimum int value is -2147483648.  
Maximum int value is 2147483647.

Where do these numbers come from?

Why Hexadecimal Numbers?

“For programmers and technicians” (Excerpt of a user manual of the chess computers *Mephisto II*, 1981)
Domain of the Type int

- Representation with $B$ bits. Domain comprises the $2^B$ integers:
  \[ \{ -2^{B-1}, -2^{B-1} + 1, \ldots, -1, 0, 1, \ldots, 2^{B-1} - 2, 2^{B-1} - 1 \} \]
- On most platforms $B = 32$
- For the type int C++ guarantees $B \geq 16$
- Background: Section 2.2.8 (Binary Representation) in the lecture notes.

Over- and Underflow

- Arithmetic operations (+, −, *) can lead to numbers outside the valid domain.
- Results can be incorrect!
  
  power8.cpp: $15^8 = -1732076671$
- There is no error message!

The Type unsigned int

- Domain
  \[ \{ 0, 1, \ldots, 2^B - 1 \} \]
- All arithmetic operations exist also for unsigned int.
- Literals: 1u, 17u ...

Mixed Expressions

- Operators can have operands of different type (e.g. int and unsigned int).
  
  \[ 17 + 17u \]
- Such mixed expressions are of the “more general” type unsigned int.
- int-operands are converted to unsigned int.
Conversion

<table>
<thead>
<tr>
<th>int Value</th>
<th>Sign</th>
<th>unsigned int Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\geq 0$</td>
<td>$x$</td>
</tr>
<tr>
<td>$x$</td>
<td>$&lt; 0$</td>
<td>$x + 2^B$</td>
</tr>
</tbody>
</table>

Due to a clever representation (two’s complement – not discussed), no addition is internally needed

Conversion “reversed”

The declaration

```c
int a = 3u;
```

converts $3u$ to $\text{int}$.

The value is preserved because it is in the domain of $\text{int}$; otherwise the result depends on the implementation.

Signed Numbers

Note: the remaining slides on signed numbers, computing with binary numbers, and the two’s complement, are not relevant for the exam

Signed Number Representation

- (Hopefully) clear by now: binary number representation without sign, e.g.

  $$[b_{31}b_{30} \ldots b_0]_u = b_{31} \cdot 2^{31} + b_{30} \cdot 2^{30} + \cdots + b_0$$

- Obviously required: use a bit for the sign.

- Looking for a consistent solution

The representation with sign should coincide with the unsigned solution as much as possible. Positive numbers should arithmetically be treated equal in both systems.
### Computing with Binary Numbers (4 digits)

#### Simple Addition

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0010</td>
<td>+</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>+0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Simple Subtraction

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0101</td>
<td>−</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>−0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Addition with Overflow

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0111</td>
<td>+</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>+1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>16</td>
<td>(1)0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Negative Numbers?

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0101</td>
<td>+</td>
<td>(−5)</td>
</tr>
<tr>
<td></td>
<td>???</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>(1)0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Simpler -1

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0001</td>
<td>+</td>
<td>(−1)</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>(1)0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Invert!

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0011</td>
<td>+</td>
<td>(−4)</td>
</tr>
<tr>
<td></td>
<td>+1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>−1</td>
<td>1111 ( \equiv 2^B - 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Utilize this:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0011</td>
<td>+</td>
<td>(−a − 1)</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>−1</td>
<td>1111 ( \equiv 2^B - 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing with Binary Numbers (4 digits)

- Negation: inversion and addition of 1
  \[-a \approx \bar{a} + 1\]

- Wrap around semantics (calculating modulo $2^B$
  \[-a \approx 2^B - a\]

Why this works

Modulo arithmetics: Compute on a circle\(^3\)

$11 \equiv 23 \equiv -1 \equiv \ldots \mod 12$

$4 \equiv 16 \equiv \ldots \mod 12$

$3 \equiv 15 \equiv \ldots \mod 12$

\(^3\)The arithmetics also work with decimal numbers (and for multiplication).

Negative Numbers (3 Digits)

<table>
<thead>
<tr>
<th>$a$</th>
<th>$-a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000 0</td>
</tr>
<tr>
<td>001</td>
<td>111 -1</td>
</tr>
<tr>
<td>010</td>
<td>110 -2</td>
</tr>
<tr>
<td>011</td>
<td>101 -3</td>
</tr>
<tr>
<td>100</td>
<td>100 -4</td>
</tr>
<tr>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>110</td>
<td>10</td>
</tr>
<tr>
<td>111</td>
<td>9</td>
</tr>
</tbody>
</table>

The most significant bit decides about the sign and it contributes to the value.

Two’s Complement

- Negation by bitwise negation and addition of 1
  \[-2 = -[0010] = [1101] + [0001] = [1110]\]

- Arithmetics of addition and subtraction identical to unsigned arithmetics
  \[3 - 2 = 3 + (-2) = [0011] + [1110] = [0001]\]

- Intuitive “wrap-around” conversion of negative numbers.
  \[-n \rightarrow 2^B - n\]

- Domain: $-2^{B-1} \ldots 2^{B-1} - 1$