2. Integers

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

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."
    return 0;
}

Precedence

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

How to put the expression in parentheses?

9 * celsius / 5 + 32

**Rule 1: precedence**

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

**From left to right**

\[ 9 \times \text{celsius} / 5 + 32 \]

deutet

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

**Rule 2: Associativity**

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

---

**Arity**

**Rule 3: Arity**

Unary operators \(\+\), \(-\) first, then binary operators \(+\), \(-\).

\[-3 - 4\]

deutet

\[ (-3) - 4 \]

---

**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

\[ 9 \times \text{celsius} / 5 + 32 \]

Order is not determined uniquely:

\[ 9 \times \text{celsius} / 5 + 32 \]

Expression Trees – Notation

Common notation: root on top

\[ 9 \times \text{celsius} / 5 + 32 \]

Evaluation Order – more formally

- Valid order: any node is evaluated \textit{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": \((a+b)\times(a++)\)
**Evaluation order**

**Guideline**

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

---

**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>Modulus</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

---

**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

\[ a = b = c \iff a = (b = c) \]

Example multiple assignment:
\[ a = b = 0 \rightarrow b=0; a=0 \]

---

**Division and Modulus**

- Operator / implements integer division
  \( 5 / 2 \) has value 2
- In `fahrenheit.cpp`
  \[ 9 \times \text{celsius} / 5 + 32 \]
  15 degrees Celsius are 59 degrees Fahrenheit
- Mathematically equivalent...but not in C++!
  \[ 9 / 5 \times \text{celsius} + 32 \]
  15 degrees Celsius are 47 degrees Fahrenheit
Division and Modulus

- Modulus-operator computes the rest of the integer division

\[ 5 \div 2 \text{ has value } 2, \quad 5 \% 2 \text{ has value } 1. \]

- It holds that:

\[(a \div b) \times b + a \% b \text{ has the value of } a.\]

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 (effects!)

In-/Decrement Operators

<table>
<thead>
<tr>
<th>Post-Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{expr}++</td>
</tr>
<tr>
<td>\text{Value of expr is increased by one, the old value of expr is returned (as R-value)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pre-Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>++\text{expr}</td>
</tr>
<tr>
<td>\text{Value of expr is increased by one, the new value of expr is returned (as L-value)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Post-Dekrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{expr}--</td>
</tr>
<tr>
<td>\text{Value of expr is decreased by one, the old value of expr is returned (as R-value)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prä-Dekrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>--\text{expr}</td>
</tr>
<tr>
<td>\text{Value of expr is increased by one, the new value of expr is returned (as L-value)}</td>
</tr>
</tbody>
</table>

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

Example

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

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)

C++ vs. ++C

Strictly speaking our language should be named ++C because

- it is an advancement of the language C
- while C++ returns the old C.

Arithmetic Assignments

```cpp
a += b

⇔

a = a + b
```

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

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

\[ b_n b_{n-1} \ldots b_1 b_0 \]

corresponds to the number \( b_n \cdot 2^n + \cdots + b_1 \cdot 2 + b_0 \)

Example: 101011 corresponds to 43.

**Binary Numbers: Numbers of the Computer?**

Truth: Computers calculate using binary numbers.

Stereotype: Computers are talking 0/1 gibberish?
Computing Tricks

Estimate the orders of magnitude of powers of two.\(^3\):

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

\(^3\)Decimal vs. binary units: MB - Megabyte vs. MiB - Megabibyte (etc.)
kilo (K, Ki) – mega (M, Mi) – giga (G, Gi) – tera (T, Ti) – peta (P, Pi) – exa (E, Ei)

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“

32-bit numbers consist of eight hex-nibbles: 0x00000000 -- 0xffffffff.

0x400 = 1Ki = 1,024.
0x10000 = 1Mi = 1,048,576.
0x40000000 = 1Gi = 1,073,741,824.

0x80000000: highest bit of a 32-bit number is set
0xffffffff: all bits of a 32-bit number are set

„0x8a20aaf0 is an address in the upper 2G of the 32-bit address space“

Example: Hex-Colors

#00FF00

r g b

Hexadecimal Numbers

Numbers with base 16

\[
h_n h_{n-1} \ldots h_1 h_0
\]

corresponds to the number

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

notation in C++: prefix 0x

Example: 0xff corresponds to 255.

Hex Nibbles

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

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

"The NZZ could have saved a lot of space ...

// Program: limits.cpp
// 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;
}

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

Where do these numbers come from?
Over- and Underflow

- Arithmetic operations (+,−,∗) can lead to numbers outside the valid domain.
- Results can be incorrect!
- There is no error message!

```cpp
power8.cpp: 15^8 = −1732076671
power20.cpp: 3^{20} = −808182895
```

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).
- 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>≥ 0</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>&lt; 0</td>
<td>x + 2^B</td>
</tr>
</tbody>
</table>

Using two complements representation, nothing happens internally.
Conversion “reversed”

The declaration
\[
\text{int } a = 3u;
\]
converts 3u to int.
The value is preserved because it is in the domain of int; otherwise the result depends on the implementation.

Signed Number Representation

(Hopefully) clear by now: binary number representation without sign, e.g.
\[
[b_{31}b_{30} \ldots b_0]_u \equiv b_{31} \cdot 2^{31} + b_{30} \cdot 2^{30} + \cdots + b_0
\]

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

\[
\begin{align*}
2 & \quad 0010 \\
+3 & \quad +0011 \\
\hline
5 & \quad 0101 \\
\end{align*}
\]

Simple Subtraction

\[
\begin{align*}
5 & \quad 0101 \\
-3 & \quad -0011 \\
\hline
2 & \quad 0010 \\
\end{align*}
\]

Addition with Overflow

\[
\begin{align*}
7 & \quad 0111 \\
+9 & \quad +1001 \\
\hline
16 & \quad (1)0000 \\
\end{align*}
\]

Negative Numbers?

\[
\begin{align*}
5 & \quad 0101 \\
+(-5) & \quad ????
\hline
0 & \quad (1)0000 \\
\end{align*}
\]
Computing with Binary Numbers (4 digits)

Simpler -1

\[
\begin{array}{c}
1 \\
+(-1) \\
\hline
0
\end{array}
\quad
\begin{array}{c}
0001 \\
1111 \\
\hline
(1)0000
\end{array}
\]

Utilize this:

\[
\begin{array}{c}
3 \\
+? \\
\hline
-1
\end{array}
\quad
\begin{array}{c}
0011 \\
+????? \\
\hline
1111
\end{array}
\]

Invert!

\[
\begin{array}{c}
3 \\
+(-4) \\
\hline
-1
\end{array}
\quad
\begin{array}{c}
0011 \\
+1100 \\
\hline
1111 \approx 2^B - 1
\end{array}
\]

Why this works

- Negation: inversion and addition of 1
  \[-a \equiv \bar{a} + 1\]
- Wrap around semantics (calculating modulo \(2^B\))
  \[-a \equiv 2^B - a\]

Modulo arithmetics: Compute on a circle\(^4\)

\[
\begin{array}{c}
11 \equiv 23 \equiv -1 \equiv \\
\ldots \mod 12
\end{array}
\quad
\begin{array}{c}
4 \equiv 16 \equiv \ldots \\
\mod 12
\end{array}
\quad
\begin{array}{c}
3 \equiv 15 \equiv \ldots \\
\mod 12
\end{array}
\]

\(^4\)The arithmetics also work with decimal numbers (and for multiplication).
Negative Numbers (4 Digits)

<table>
<thead>
<tr>
<th>( a )</th>
<th>(-a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000 000</td>
</tr>
<tr>
<td>1</td>
<td>001 111</td>
</tr>
<tr>
<td>2</td>
<td>010 110</td>
</tr>
<tr>
<td>3</td>
<td>011 101</td>
</tr>
<tr>
<td>4</td>
<td>100 100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

The most significant bit decides about the sign.

Two’s Complement

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

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

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

- Domain: \(-2^{B-1} \ldots 2^{B-1} - 1\)