1. Integers

Evaluation of Arithmetic Expressions, Associativity and Precedence, Arithmetic Operators, Domain of Types \texttt{int}, \texttt{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.\n";
    return 0;
}

15 degrees Celsius are 59 degrees Fahrenheit

---

**Precedence**

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

How to put the expression in parentheses?

- Multiplication/Division before Addition/Subtraction

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

bedeutet

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

**Rule 1: precedence**

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

From left to right

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

bedeutet

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

**Rule 2: Associativity**

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

**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).

**Arity**

**Rule 3: Arity**

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

\[-3 - 4\]

means

\[ (-3) - 4 \]

**Expression Trees**

Parentheses yield the expression tree

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

\[ 9 \quad \text{celsius} \quad 5 \quad 32 \]

\[ + \quad / \quad * \]

5

7
**Evaluation Order**

"From top to bottom" in the expression tree

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

- Order is not determined uniquely:

**Expression Trees – Notation**

Common notation: root on top

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

**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": \((a+b) \times (a++)\)
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>

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:
\( a = b = 0 \iff b=0; a=0 \)

Division and Modulus

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

- Modulus-operator computes the rest of the integer division
  \[ \frac{5}{2} \text{ has value 2, } 5 \% 2 \text{ has value 1.} \]

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

### 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

- **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 increased by one, the new value of expr is returned (as L-value)

<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</td>
</tr>
<tr>
<td>Pre-increment</td>
<td>++expr</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>Post-decrement</td>
<td>expr--</td>
<td>1</td>
<td>17</td>
<td>left</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>--expr</td>
<td>1</td>
<td>16</td>
<td>right</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
```

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\})

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

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

Example: 101011 corresponds to 43.

- **Most Significant Bit (MSB)**
- **Least Significant Bit (LSB)**

### Binary Numbers: Numbers of the Computer?

**Truth:** Computers calculate using binary numbers.

**Stereotype:** Computers are talking 0/1 gibberish

![Image of computer and people working on a machine]

![Image of a newspaper article about computer science]
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.

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.
0x100000 = 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

Why Hexadecimal Numbers?

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

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

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: $15^8 = -1732076671$
  - power20.cpp: $3^{20} = -808182895$
- There is no error message!

Domain of Type int

// 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.
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 \geq 0 )</td>
<td></td>
<td>( x )</td>
</tr>
<tr>
<td>( x &lt; 0 )</td>
<td></td>
<td>( x + 2^B )</td>
</tr>
</tbody>
</table>

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.

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

- 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.