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.\n";
  return 0;
}
```


`9 * celsius / 5 + 32`

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

How to put the expression in parentheses?

Precedence

Multiplication/Division before Addition/Subtraction

`9 * celsius / 5 + 32`

bedeutet

`(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 * celsius / 5 + 32`

bedeutet

`((9 * 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`

means

`(-3) - 4`

Parentheses

Any expression can be put in parentheses by means of

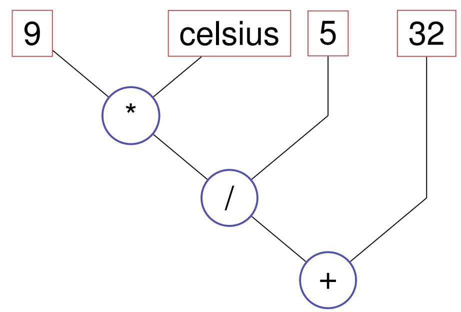
- associativities
- precedences
- arities (number of operands)

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

Expression Trees

Parentheses yield the expression tree

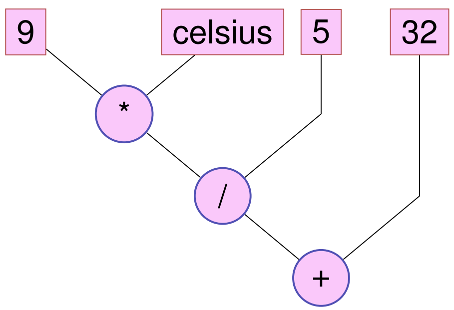
`(((9 * celsius) / 5) + 32)`



Evaluation Order

"From top to bottom" in the expression tree

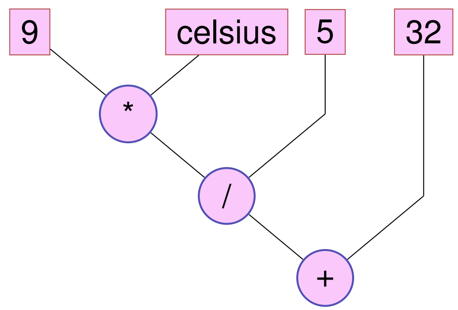
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

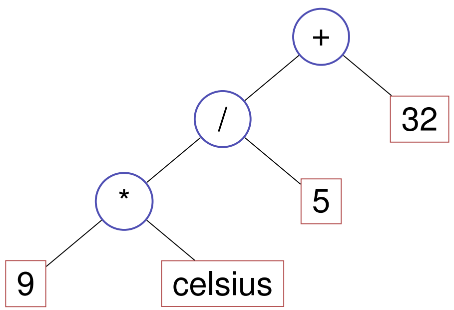
```
9 * celsius / 5 + 32
```



Expression Trees – Notation

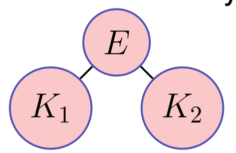
Common notation: root on top

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



Evaluation Order – more formally

- Valid order: any node is evaluated *after* its children



In $\mathrm{C{+}{+}}$, 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)`

Evaluation order

Guideline

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

Arithmetic operations

	Symbol	Arity	Precedence	Associativity
Unary +	+	1	16	right
Negation	−	1	16	right
Multiplication	*	2	14	left
Division	/	2	14	left
Modulo	%	2	14	links
Addition	+	2	13	left
Subtraction	−	2	13	left

All operators: [R-value $\times$] R-value $\to$ 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` $\iff$ `a = (b = c)`

Example multiple assignment:
`a = b = 0` $\Longrightarrow$ `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

`5 / 2` has value 2, `5 % 2` has value 1.

- It holds that:

`(a / b) * b + a % b` 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
  - 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 increased by one, the *new* value of `expr` is returned (as L-value)

In-/decrement Operators

	use	arity	prec	assoz	L-/R-value
Post-increment	`expr++`	1	17	left	L-value → R-value
Pre-increment	`++expr`	1	16	right	L-value → L-value
Post-decrement	`expr--`	1	17	left	L-value → R-value
Pre-decrement	`--expr`	1	16	right	L-value → L-value

In-/Decrement Operators

Example

```
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)

`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

$$\texttt{a += b}$$
$$\Leftrightarrow$$
$$\texttt{a = a + b}$$

analogously for `-`, `*`, `/` and `%`

Arithmetic Assignments

	Gebrauch	Bedeutung
`+=`	`expr1 += expr2`	`expr1 = expr1 + expr2`
`-=`	`expr1 -= expr2`	`expr1 = expr1 - expr2`
`*=`	`expr1 *= expr2`	`expr1 = expr1 * expr2`
`/=`	`expr1 /= expr2`	`expr1 = expr1 / expr2`
`%=`	`expr1 %= expr2`	`expr1 = expr1 % expr2`

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

Binary Number Representations

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

$$b_n b_{n-1} \dots b_1 b_0$$

corresponds to the number $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

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]:

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

[2] 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)

Hexadecimal Numbers

Numbers with base 16

$$h_n h_{n-1} \dots h_1 h_0$$

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

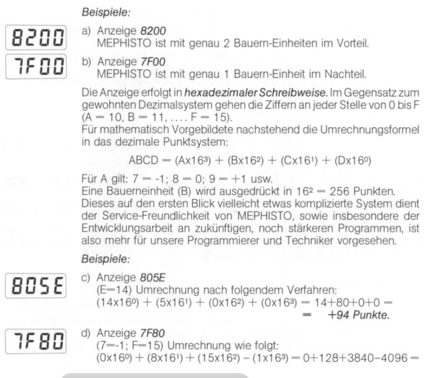
hex	bin	dec
0	0000	0
1	**0001**	**1**
2	**0010**	**2**
3	0011	3
4	**0100**	**4**
5	0101	5
6	0110	6
7	0111	7
8	**1000**	**8**
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

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”

Why Hexadecimal Numbers?

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



Beispiele:

8200 a) Anzeige *8200*
MEPHISTO ist mit genau 2 Bauern-Einheiten im Vorteil.

7F00 b) Anzeige *7F00*
MEPHISTO ist mit genau 1 Bauern-Einheit im Nachteil.

Die Anzeige erfolgt in ***hexadezimaler Schreibweise.*** Im Gegensatz zum gewohnten Dezimalsystem gehen die Ziffern an jeder Stelle von 0 bis F (A = 10, B = 11, ... F = 15).
Für mathematisch Vorgebildete nachstehend die Umrechnungsformel in das dezimale Punktsystem:

$$ABCD = (A \times 16^3) + (B \times 16^2) + (C \times 16^1) + (D \times 16^0)$$

Für A gilt: 7 = -1; 8 = 0; 9 = +1 usw.
Eine Bauerneinheit (B) wird ausgedrückt in $16^2 = 256$ Punkten.
Dieses auf den ersten Blick vielleicht etwas komplizierte System dient der Service-Freundlichkeit von MEPHISTO, sowie insbesondere der Entwicklungsarbeit an zukünftigen, noch stärkeren Programmen, ist also mehr für unsere Programmierer und Techniker vorgesehen.

Beispiele:

805E c) Anzeige *805E*
(E=14) Umrechnung nach folgendem Verfahren:
$(14 \times 16^0) + (5 \times 16^1) + (0 \times 16^2) + (0 \times 16^3) = 14+80+0+0 =$
$=$ *+94 Punkte.*

7F80 d) Anzeige *7F80*
(7=-1; F=15) Umrechnung wie folgt:
$(0 \times 16^0) + (8 \times 16^1) + (15 \times 16^2) - (1 \times 16^3) = 0+128+3840-4096 =$

Example: Hex-Colors

$$\#\underbrace{00}_{r}\underbrace{FF}_{g}\underbrace{00}_{b}$$

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?

Domain of the Type `int`

- Representation with B bits. Domain comprises the 2^B integers:

$$\{-2^{B-1}, -2^{B-1}+1, \ldots, -1, 0, 1, \ldots, 2^{B-1}-2, 2^{B-1}-1\}$$

Where does this partitioning come from?

- On most platforms $B = 32$
- For the type **`int`** $\mathrm{C}{++}$ 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!*

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

`int` Value	Sign	`unsigned int` Value
x	≥ 0	x
x	< 0	$x + 2^B$

Using two’s complement representation (to come), nothing happens internally

Conversion “reversed”

The declaration

```
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.

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

- 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

$$\begin{array}{r} 2 \\ +3 \\ \hline 5 \end{array} \qquad \begin{array}{r} 0010 \\ +0011 \\ \hline 0101 \end{array}$$

Simple Subtraction

$$\begin{array}{r} 5 \\ -3 \\ \hline 2 \end{array} \qquad \begin{array}{r} 0101 \\ -0011 \\ \hline 0010 \end{array}$$

Computing with Binary Numbers (4 digits)

Addition with Overflow

$$\begin{array}{r} 7 \\ +9 \\ \hline 16 \end{array} \qquad \begin{array}{r} 0111 \\ +1001 \\ \hline (1)0000 \end{array}$$

Negative Numbers?

$$\begin{array}{r} 5 \\ +(-5) \\ \hline 0 \end{array} \qquad \begin{array}{r} 0101 \\ ???? \\ \hline (1)0000 \end{array}$$

Computing with Binary Numbers (4 digits)

Simpler -1

$$\begin{array}{r} 1 \\ +(-1) \\ \hline 0 \end{array} \qquad \begin{array}{r} 0001 \\ 1111 \\ \hline (1)0000 \end{array}$$

Utilize this:

$$\begin{array}{r} 3 \\ +? \\ \hline -1 \end{array} \qquad \begin{array}{r} 0011 \\ +???? \\ \hline 1111 \end{array}$$

Computing with Binary Numbers (4 digits)

Invert!

$$\begin{array}{r} 3 \\ +(-4) \\ \hline -1 \end{array} \qquad \begin{array}{r} 0011 \\ +1100 \\ \hline 1111 \end{array} \widehat{=}\ 2^B - 1$$

$$\begin{array}{r} a \\ +(-a-1) \\ \hline -1 \end{array} \qquad \begin{array}{r} a \\ \bar{a} \\ \hline 1111 \end{array} \widehat{=}\ 2^B - 1$$

Computing with Binary Numbers (4 digits)

- Negation: inversion and addition of 1

$$-a \;\widehat{=}\; \bar{a} + 1$$

- Wrap around semantics (calculating modulo 2^B

$$-a \;\widehat{=}\; 2^B - a$$

Why this works

Modulo arithmetics: Compute on a circle[3]

$11 \equiv 23 \equiv -1 \equiv \ldots \mod 12$ $\qquad 4 \equiv 16 \equiv \ldots \mod 12$ $\qquad 3 \equiv 15 \equiv \ldots \mod 12$

[3]The arithmetics also work with decimal numbers (and for multiplication).

Negative Numbers (3 Digits)

	a	$-a$	
0	000	000	0
1	001	**111**	-1
2	010	**110**	-2
3	011	**101**	-3
4	100	**100**	-4
5	101		
6	110		
7	111		

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.

$$-n \to 2^B - n$$

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