

3. 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;
}
```

110

111

`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 (`+`, `-`)

112

113

Associativity

From left to right

$9 * \text{celsius} / 5 + 32$

bedeutet

$((9 * \text{celsius}) / 5) + 32$

Rule 2: Associativity

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

114

Arity

Rule 3: Arity

Unary operators $+$, $-$ first, then binary operators $+$, $-$.

$-3 - 4$

means

$(-3) - 4$

115

Parentheses

Any expression can be put in parentheses by means of

- associativities
- precedences
- arities (number of operands)

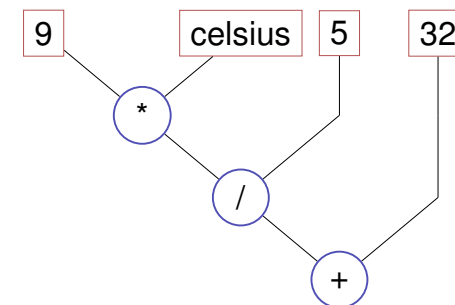
of the operands in an unambiguous way (Details in the lecture notes).

116

Expression Trees

Parentheses yield the expression tree

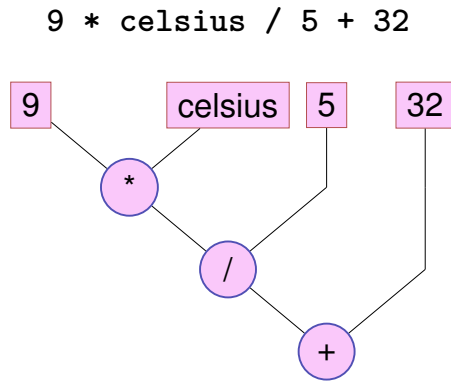
$((9 * \text{celsius}) / 5) + 32$



117

Evaluation Order

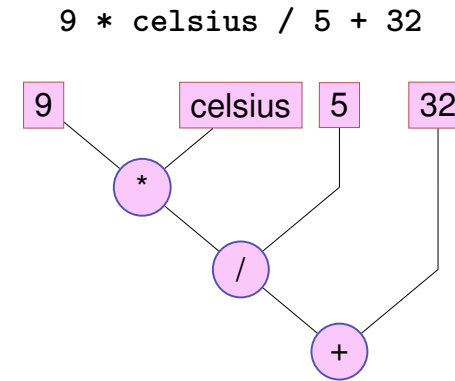
"From top to bottom" in the expression tree



118

Evaluation Order

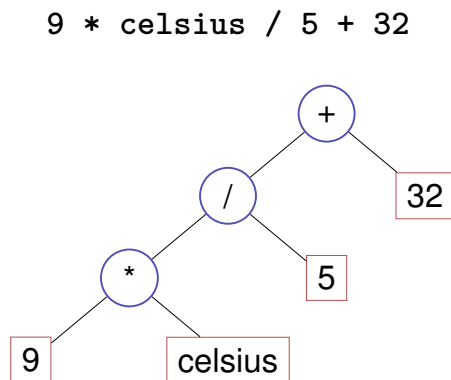
Order is not determined uniquely:



119

Expression Trees – Notation

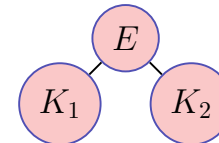
Common notation: root on top



120

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": $(a+b)*(a++)$

121

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
Modulus	%	2	14	links
Addition	+	2	13	left
Subtraction	-	2	13	left

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

122

123

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` \implies `b=0; a=0`

124

Division and Modulus

- 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

125

Division and Modulus

- Modulus-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`.

126

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!)

127

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-Decrement

`expr--`

Value of `expr` is decreased by one, the *old* value of `expr` is returned (as R-value)

Prä-Decrement

`--expr`

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

128

In-/decrement Operators

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

129

In-/Decrement Operators

Example

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

130

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)

131

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.

132

Arithmetic Assignments

`a += b`

⇔

`a = a + b`

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

133

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.

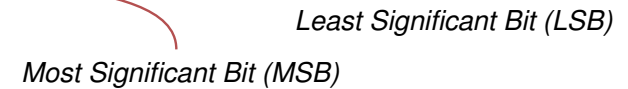
Binary Number Representations

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.



Binary Numbers: Numbers of the Computer?

Truth: Computers calculate using binary numbers.



Binary Numbers: Numbers of the Computer?

Stereotype: computers are talking 0/1 gibberish



Computing Tricks

- Estimate the orders of magnitude of powers of two.³:

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

³Decimal 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)

138

Hexadecimal Numbers

Numbers with base 16

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

corresponds to the number

$$h_n \cdot 16^n + \dots + h_1 \cdot 16 + h_0.$$

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

139

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”

140

Example: Hex-Colors

#00FF00
r g b

141

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 Umrechnungsmel 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 Bauerninheit (B) wird ausgedrückt in 16² = 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:
(14x16³) + (5x16²) + (0x16¹) + (0x16⁰) = 14+80+0+0 = +94 Punkte.

7F80 d) Anzeige 7F80
(7=-1; F=15) Umrechnung wie folgt:
(0x16³) + (8x16²) + (15x16¹) - (1x16⁰) = 0+128+384-4096 =

http://www.zanchetta.net/default.aspx?Category=ECHLIQUERS&Page=documentations 142

Why Hexadecimal Numbers?

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

143

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.

Where do these numbers come from?

144

Domain of the Type int

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

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

- 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.

Where does this partitioning come from?

145

Over- and Underflow

- Arithmetic operations (+, -, *) can lead to numbers outside the valid domain.
- Results can be incorrect!

```
power8.cpp: 158 = -1732076671
```

```
power20.cpp: 320 = -808182895
```

- There is *no error message!*

146

The Type unsigned int

- Domain

$$\{0, 1, \dots, 2^B - 1\}$$

- All arithmetic operations exist also for `unsigned int`.
- Literals: `1u`, `17u` ...

147

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`.

148

Conversion

int Value	Sign	unsigned int Value
-----------	------	--------------------

x	≥ 0	x
x	< 0	$x + 2^B$

Using two complements representation, nothing happens internally

149

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.

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

- 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.

150

151

Computing with Binary Numbers (4 digits)

Simple Addition

2	0010
+3	+0011
-----	-----
5	0101

Simple Subtraction

5	0101
-3	-0011
-----	-----
2	0010

152

Computing with Binary Numbers (4 digits)

Addition with Overflow

7	0111
+9	+1001
-----	-----
16	(1)0000

Negative Numbers?

5	0101
+(-5)	????
-----	-----
0	(1)0000

153

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!

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

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

154

155

Computing with Binary Numbers (4 digits)

- Negation: inversion and addition of 1

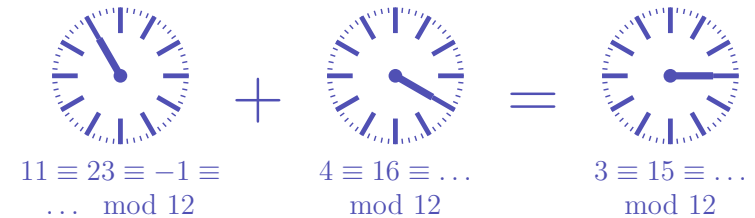
$$-a \hat{=} \bar{a} + 1$$

- Wrap around semantics (calculating modulo 2^B)

$$-a \hat{=} 2^B - a$$

Why this works

Modulo arithmetics: Compute on a circle⁴



⁴The arithmetics also work with decimal numbers (and for multiplication).

156

157

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.

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: $-2^{B-1} \dots 2^{B-1} - 1$