2. Integers

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

Example: power8.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';
```

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;
}
```

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`

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

- Arithmetic expression,

`9 * celsius / 5 + 32`

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

- Arithmetic expression,
- three literals, one variable, three operator symbols

`9 * celsius / 5 + 32`

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

- Arithmetic expression,
- three literals, one variable, three operator symbols

`9 * celsius / 5 + 32`

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

- Arithmetic expression,
- three literals, one variable, three operator symbols

`9 * celsius / 5 + 32`

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

- Arithmetic expression,
- three literals, one 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
```

Precedence

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
```

Associativity

Rule 2: Associativity

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

Arity

Sign

```
-3 - 4
```

means

```
(-3) - 4
```

Arity

Rule 3: Arity

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

Parentheses

Any expression can be put in parentheses by means of

- associativities
- precedences
- arities

of the operands in an unambiguous way.

Expression Trees

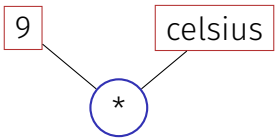
Parentheses yield the expression tree

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

Expression Trees

Parentheses yield the expression tree

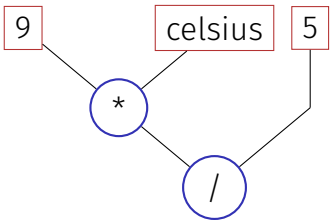
```
(9 * celsius) / 5 + 32
```



Expression Trees

Parentheses yield the expression tree

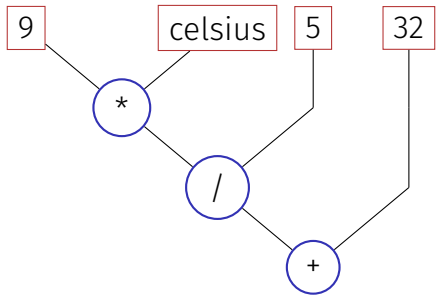
```
((9 * celsius) / 5) + 32
```



Expression Trees

Parentheses yield the expression tree

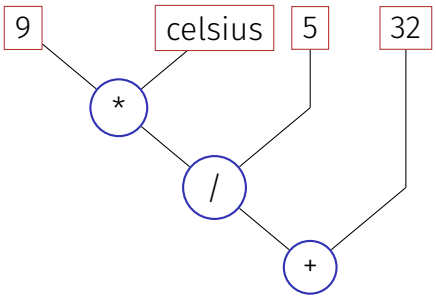
```
(((9 * celsius) / 5) + 32)
```



Evaluation Order

"From top to bottom" in the expression tree

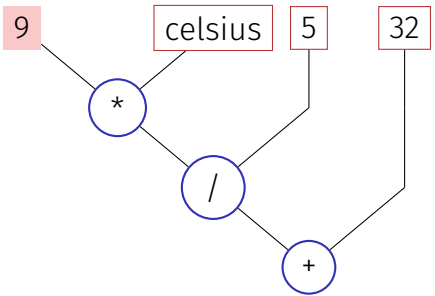
```
9 * celsius / 5 + 32
```



Evaluation Order

"From top to bottom" in the expression tree

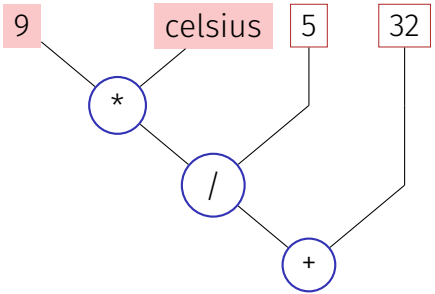
```
9 * celsius / 5 + 32
```



Evaluation Order

"From top to bottom" in the expression tree

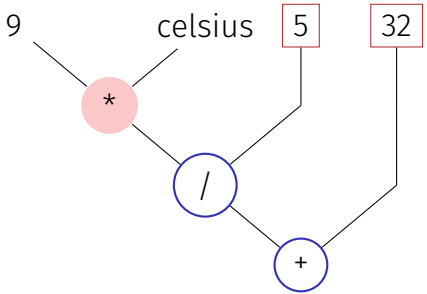
```
9 * celsius / 5 + 32
```



Evaluation Order

"From top to bottom" in the expression tree

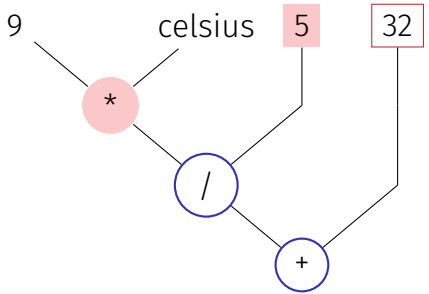
```
9 * celsius / 5 + 32
```



Evaluation Order

"From top to bottom" in the expression tree

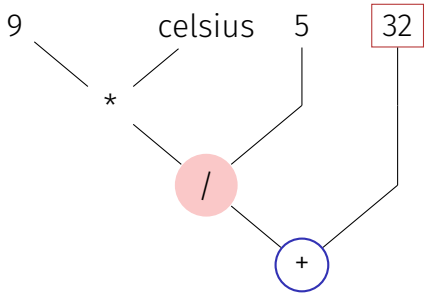
```
9 * celsius / 5 + 32
```



Evaluation Order

"From top to bottom" in the expression tree

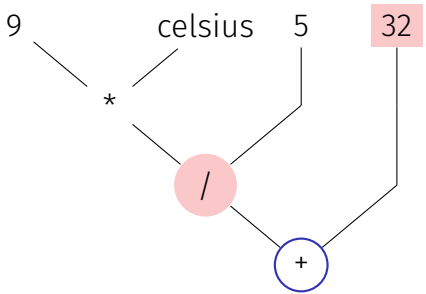
```
9 * celsius / 5 + 32
```



Evaluation Order

"From top to bottom" in the expression tree

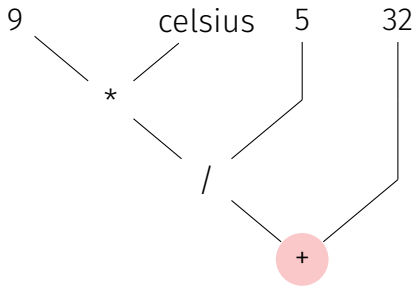
```
9 * celsius / 5 + 32
```



Evaluation Order

"From top to bottom" in the expression tree

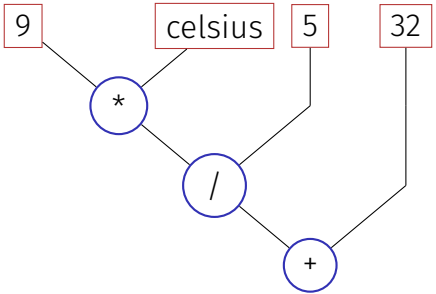
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

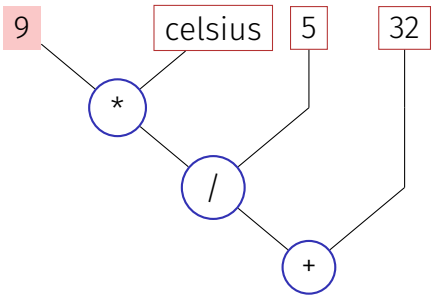
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

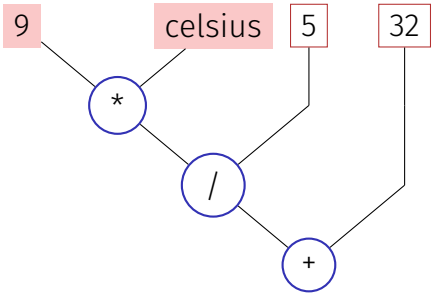
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

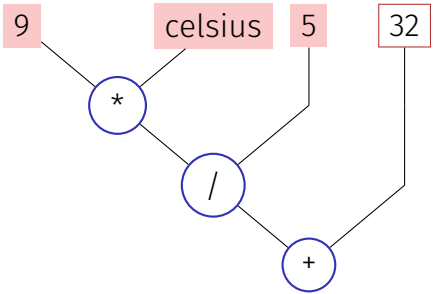
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

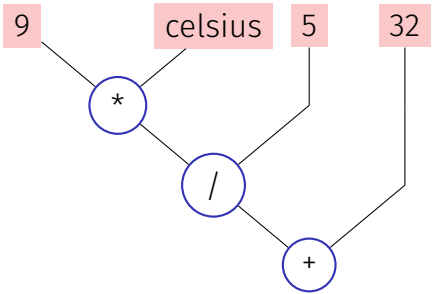
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

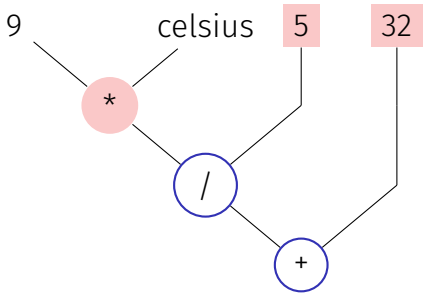
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

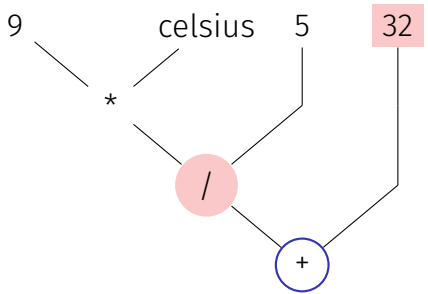
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

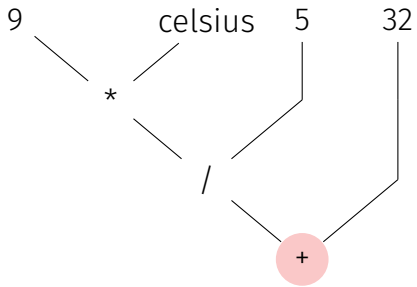
```
9 * celsius / 5 + 32
```



Evaluation Order

Order is not determined uniquely:

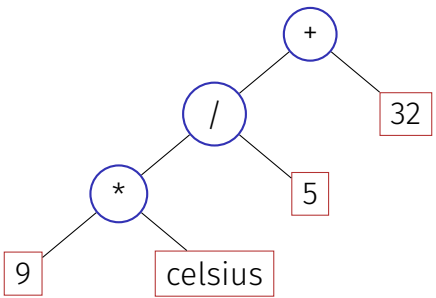
```
9 * celsius / 5 + 32
```



Expression Trees – Notation

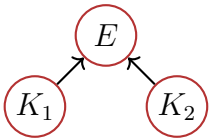
Common notation: root on top

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



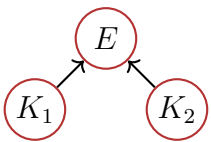
Evaluation Order – more formally

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



Evaluation Order – more formally

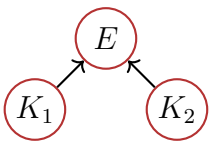
Valid order: any node is evaluated **after** its children



C++: the valid order to be used is not defined.

Evaluation Order – more formally

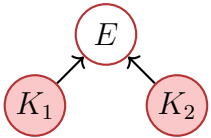
Valid order: any node is evaluated **after** its children



C++: the valid order to be used is not defined.

Evaluation Order – more formally

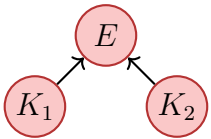
Valid order: any node is evaluated **after** its children



C++: the valid order to be used is not defined.

Evaluation Order – more formally

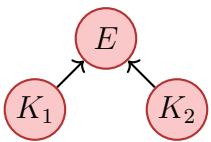
Valid order: any node is evaluated **after** its children



C++: the valid order to be used is not defined.

Evaluation Order – more formally

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

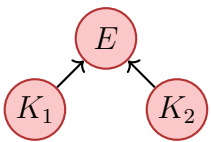


C++: the valid order to be used is not defined.

- "Good expression": any valid evaluation order leads to the same result.

Evaluation Order – more formally

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



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

Interlude: Assignment expression – in more detail

- Already known: `a = b` means Assignment of `b` (R-value) to `a` (L-value). Returns: L-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?

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

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

$$\texttt{a = b = c} \iff \texttt{a = (b = c)}$$

Multiple assignment: `a = b = 0` $\Longrightarrow$ `b=0; a=0`

Division

- Operator `/` implements integer division

  `5 / 2` has value 2

Division

- Operator `/` implements integer division

  `5 / 2` has value 2

- In `fahrenheit.cpp`

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

```
15 degrees Celsius are 59 degrees Fahrenheit
```

Division

- Operator `/` implements integer division

  `5 / 2` has value 2

- In `fahrenheit.cpp`

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

```
15 degrees Celsius are 59 degrees Fahrenheit
```

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…

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

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…

```
1 * celsius + 32
```

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…

```
15 + 32
```

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…

```
47
```

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 == -(a/b)
```

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 == -(a/b)
```

- It also holds:

  `(a / b) * b + a % b` has the value of `a`.

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 == -(a/b)
  ```

- It also holds:

  `(a / b) * b + a % b` 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.
```

Increment and decrement

```
expr = expr + 1.
```

Disadvantages

- relatively long

Increment and decrement

```
expr = expr + 1.
```

Disadvantages

- relatively long
- **expr** is evaluated twice
  - Later: L-valued expressions whose evaluation is “expensive”

Increment and decrement

```
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)

In-/Decrement Operators

Pre-increment

```
++expr
```

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

In-/Decrement Operators

Post-Dekrement

```
expr--
```

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

In-/Decrement Operators

Prä-Dekrement

```
--expr
```

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

In-/Decrement Operators

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

In-/Decrement Operators

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

In-/Decrement Operators

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

In-/Decrement Operators

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

Arithmetic Assignments

```
a += b
```

$\Leftrightarrow$

```
a = a + b
```

Arithmetic Assignments

```
a += b
```

$\Leftrightarrow$

```
a = a + b
```

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

Binary Number Representations

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

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

corresponds to the number $b_n \cdot 2^n + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$

Binary Number Representations

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

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

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

Binary Number Representations

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

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

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

```
101011
```

Binary Number Representations

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

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

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

101011 corresponds to **32+8+2+1**.

Binary Number Representations

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

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

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

101011 corresponds to **43**.

Binary Number Representations

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

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

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

101011 corresponds to **43**.

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`**

`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

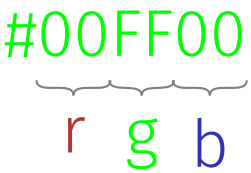
Why Hexadecimal Numbers?

- A Hex-Nibble requires exactly 4 bits.

Why Hexadecimal Numbers?

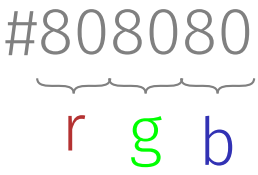
- A Hex-Nibble requires exactly 4 bits.
- “compact representation of binary numbers”

Example: Hex-Colors



Example: Hex-Colors

Example: Hex-Colors



Example: Hex-Colors

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;
}
```

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.
```

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

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

Domain of the Type `int`

- Representation with B bits. Domain

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

- On most platforms $B = 32$

Domain of the Type `int`

- Representation with B bits. Domain

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

- For the type **`int`** C++ guarantees $B \geq 16$

Over- and Underflow

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

  `power8.cpp`: $15^8 = -1732076671$

- In C++, over-/underflow of **`int`** are undefined behaviour, i.e. no error messages, not guarantees

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	$x + 2^B$

Conversion

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

Due to a clever representation (two's complement), no addition is internally needed

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.

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

- 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

$$\begin{array}{r} 2 \\ +3 \\ \hline 5 \end{array} \qquad \begin{array}{r} 0010 \\ +0011 \\ \hline 0101_2 \end{array} \quad = 5_{10}$$

Computing with Binary Numbers (4 digits)

Simple Subtraction

$$\begin{array}{r} 5 \\ -3 \\ \hline 2 \end{array} \qquad \begin{array}{r} 0101 \\ -0011 \\ \hline 0010_2 \end{array} \quad = 2_{10}$$

Computing with Binary Numbers (4 digits)

Addition with Overflow

$$\begin{array}{rr} 7 & 0111 \\ +10 & +1010 \\ \hline 17 & (1)0001_2 \end{array} \quad = 1_{10} (= 17 \bmod 16)$$

Computing with Binary Numbers (4 digits)

Subtraction with underflow

$$\begin{array}{rr} 5 & 0101 \\ +(-10) & 1010 \\ \hline -5 & (\ldots 11)1011_2 \end{array} \quad = 11_{10} (= -5 \bmod 16)$$

Why this works

Modulo arithmetics: Compute on a circle[2]

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

Negative Numbers (3 Digits)

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

Negative Numbers (3 Digits)

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

Negative Numbers (3 Digits)

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

Negative Numbers (3 Digits)

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

Negative Numbers (3 Digits)

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

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		

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		

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.