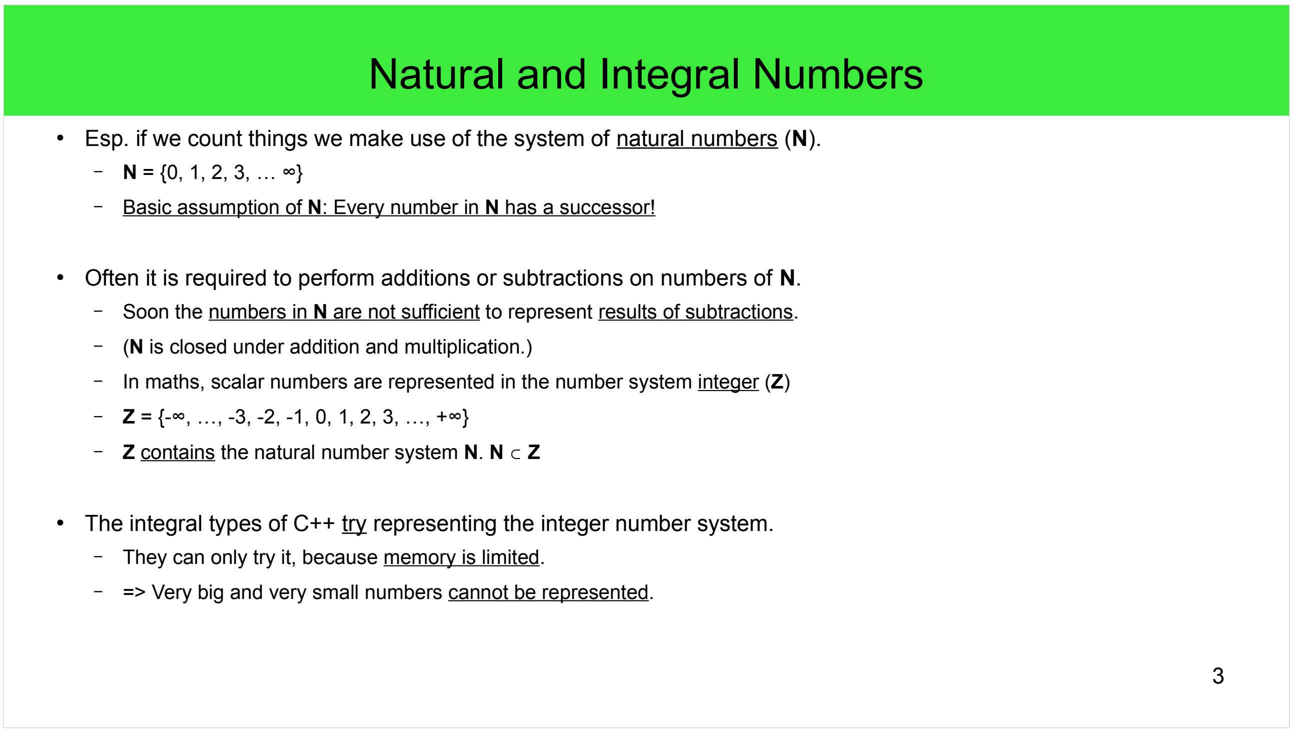
(7) Basics of the C++ Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

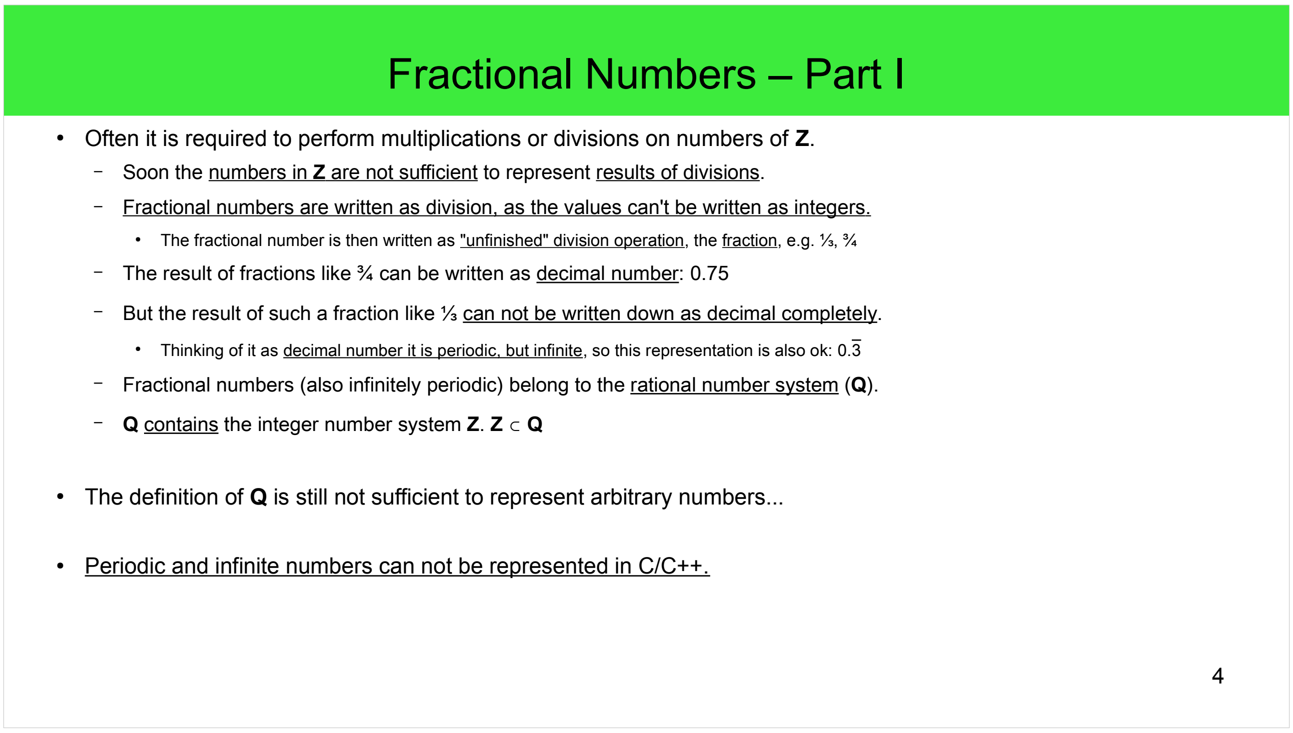
- Sources:
  - Bjarne Stroustrup, The C++ Programming Language
  - Charles Petzold, Code
  - Tom Wendel, MSDN Webcast: Mathe für Programmierer
  - Richard Knerr, Goldmann Lexikon Mathematik
  - Rob Williams, Computer System Architecture
  - Jerry Cain, Stanford Course CS 107



Natural and Integral Numbers

- Esp. if we count things we make use of the system of natural numbers (**N**).
  - **N** = {0, 1, 2, 3, … ∞}
  - Basic assumption of **N**: Every number in **N** has a successor!
- Often it is required to perform additions or subtractions on numbers of **N**.
  - Soon the numbers in **N** are not sufficient to represent results of subtractions.
  - (**N** is closed under addition and multiplication.)
  - In maths, scalar numbers are represented in the number system integer (**Z**)
  - **Z** = {-∞, …, -3, -2, -1, 0, 1, 2, 3, …, +∞}
  - **Z** contains the natural number system **N**. **N** ⊂ **Z**
- The integral types of C++ try representing the integer number system.
  - They can only try it, because memory is limited.
  - => Very big and very small numbers cannot be represented.

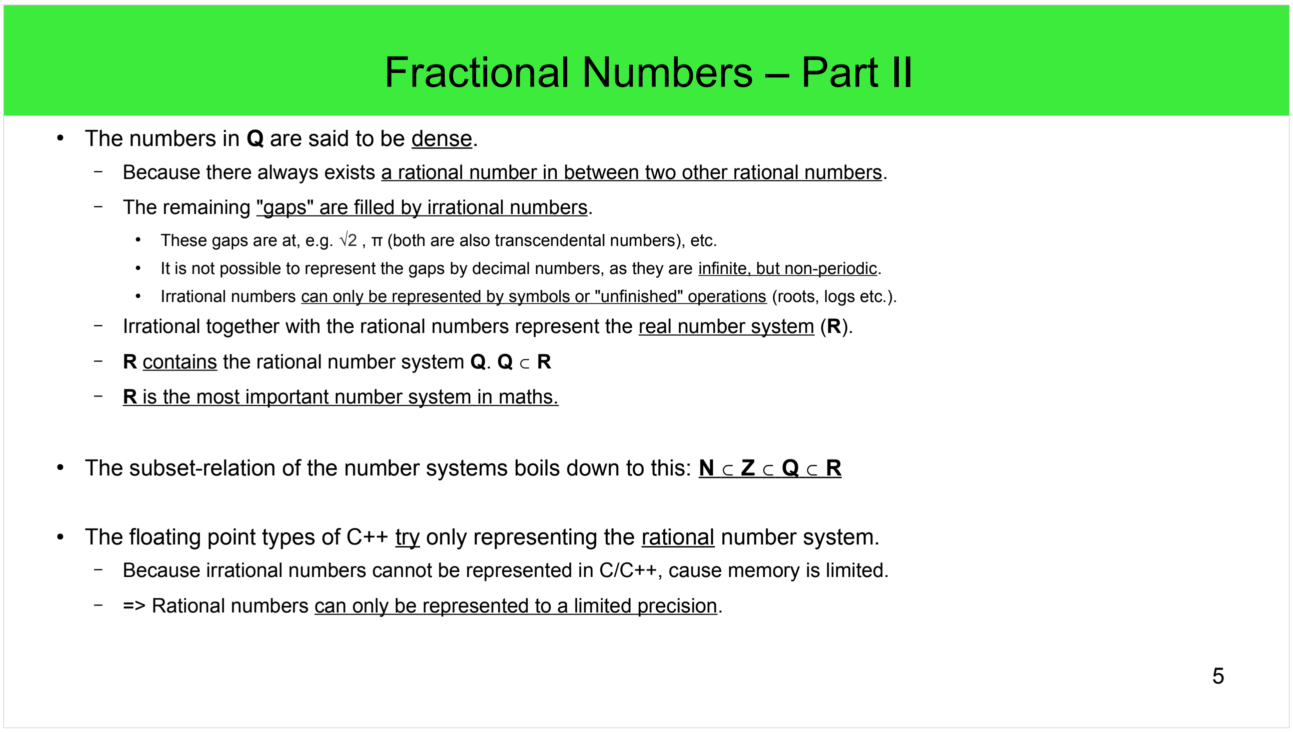
- **N**'s closure means that the addition and the multiplication of natural numbers results in natural numbers.



Fractional Numbers – Part I

- Often it is required to perform multiplications or divisions on numbers of **Z**.
  - Soon the numbers in **Z** are not sufficient to represent results of divisions.
  - Fractional numbers are written as division, as the values can't be written as integers.
    - The fractional number is then written as "unfinished" division operation, the fraction, e.g. ⅓, ¾
  - The result of fractions like ¾ can be written as decimal number: 0.75
  - But the result of such a fraction like ⅓ can not be written down as decimal completely.
    - Thinking of it as decimal number it is periodic, but infinite, so this representation is also ok: $0.\overline{3}$
  - Fractional numbers (also infinitely periodic) belong to the rational number system (**Q**).
  - **Q** contains the integer number system **Z**. $\mathbf{Z} \subset \mathbf{Q}$
- The definition of **Q** is still not sufficient to represent arbitrary numbers...
- Periodic and infinite numbers can not be represented in C/C++.

- Meanwhile, negative numbers are also kind of "natural" to us. – Just mind debts, which are a kind of negative asset!



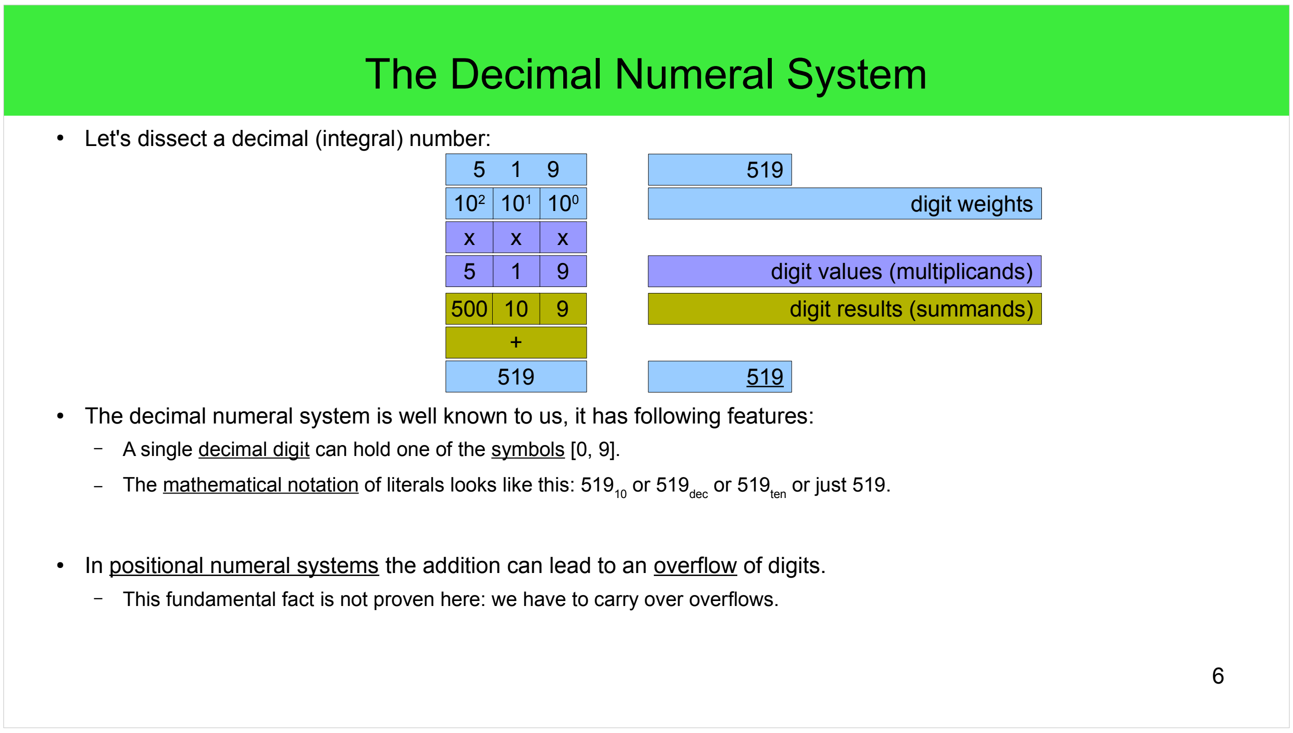
Fractional Numbers – Part II

- The numbers in **Q** are said to be dense.
  - Because there always exists a rational number in between two other rational numbers.
  - The remaining "gaps" are filled by irrational numbers.
    - These gaps are at, e.g. √2 , π (both are also transcendental numbers), etc.
    - It is not possible to represent the gaps by decimal numbers, as they are infinite, but non-periodic.
    - Irrational numbers can only be represented by symbols or "unfinished" operations (roots, logs etc.).
  - Irrational together with the rational numbers represent the real number system (**R**).
  - **R** contains the rational number system **Q**. $\mathbf{Q} \subset \mathbf{R}$
  - **R** is the most important number system in maths.

- The subset-relation of the number systems boils down to this: $\mathbf{N} \subset \mathbf{Z} \subset \mathbf{Q} \subset \mathbf{R}$

- The floating point types of C++ try only representing the rational number system.
  - Because irrational numbers cannot be represented in C/C++, cause memory is limited.
  - => Rational numbers can only be represented to a limited precision.

- Transcendental numbers are those irrational numbers that can not be represented as a solution of a polynomial expression having only rational coefficients. Transcendental numbers are real numbers.
- There are still operations that can not be solved in **R**:
  - The solution of √-1 can not be solved in **R**. It can, however, be solved in the complex number system **C**.
  - The division by zero is generally not defined.
- => Very big and very small numbers can be represented with floating point types, but some limits are still present. The precision is the significant problem here.



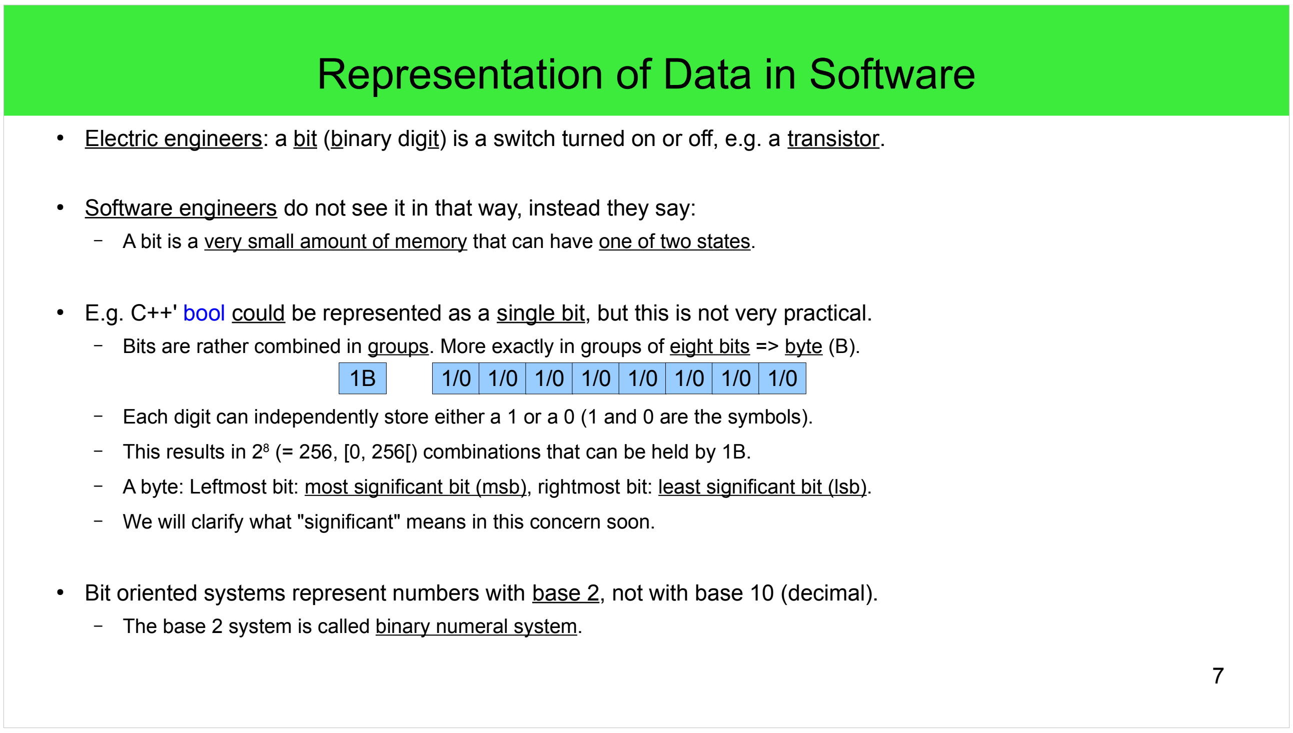
The Decimal Numeral System

- Let's dissect a decimal (integral) number:

5	1	9	519
10^2	10^1	10^0	digit weights
x	x	x	
5	1	9	digit values (multiplicands)
500	10	9	digit results (summands)
+			
519			519

- The decimal numeral system is well known to us, it has following features:
  - A single decimal digit can hold one of the symbols [0, 9].
  - The mathematical notation of literals looks like this: 519_{10} or 519_{dec} or 519_{ten} or just 519.
- In positional numeral systems the addition can lead to an overflow of digits.
  - This fundamental fact is not proven here: we have to carry over overflows.

- Why do we use the decimal system in our everyday life?
  - Because we used our ten fingers to count things for ages.
  - The word "digit" is taken from the latin word "digitus" for "finger".
- The decimal system was introduced by Arabian mathematicians. It introduces the idea of the "zero", a basic invention allowing algebra "to work" (algebra was also introduced by the Arabians). Interestingly it has no symbol for the ten, which all former systems had. The new idea of having "zero" is, that it has its own symbol as peer among other numbers.
- There also exist non- or partial-positional numeral systems like the roman numeral system, which is a sign-value sum system.
  - In a sum system the position of digits is not directly significant (the roman system partially positional). Well, such systems are good to count things, but calculations with those systems is hard. Maybe because of this, the roman empire wasn't a source of great mathematicians.



Representation of Data in Software

- Electric engineers: a bit (binary digit) is a switch turned on or off, e.g. a transistor.
- Software engineers do not see it in that way, instead they say:
  - A bit is a very small amount of memory that can have one of two states.
- E.g. C++' bool could be represented as a single bit, but this is not very practical.
  - Bits are rather combined in groups. More exactly in groups of eight bits => byte (B).

1B	1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0

  - Each digit can independently store either a 1 or a 0 (1 and 0 are the symbols).
  - This results in 2^8 (= 256, [0, 256[) combinations that can be held by 1B.
  - A byte: Leftmost bit: most significant bit (msb), rightmost bit: least significant bit (lsb).
  - We will clarify what "significant" means in this concern soon.
- Bit oriented systems represent numbers with base 2, not with base 10 (decimal).
  - The base 2 system is called binary numeral system.

- In soft- and hardware the internal representation is not done in the decimal numeral system, because values need to be represented based on an electrical platform.
- Electric engineers: also high and low voltages can represent the state of 1b, because the system can be based on currents within a range. Bit storage can be implemented as flip-flop (a flip-flop is an electronic circuit that can remember a state of 1b) in transistor-transistor logic (TTL, old, needs much power, e.g. with two cascaded NOR-gates, as "level-trigger-D-type-latch") or complementary metal-oxide-semiconductor (CMOS, newer – the predominant implementation today, needs lesser power, high density possible (e.g. very large-scale integration (VSLI) and even more complex)).
- Since the multiplicators of the positions each can only be 1 or 0, the calculation of the sum is even easier than in the decimal numeral system. – The 0 multiplications simply fall down.
- When do we use the notations bit (b) and byte (B)?
  - Bits are used to define bit-patterns, signal encodings and the bitrate of serial communication (USB (5Gbit/s), LAN (1Gibit/s), Modem (56Kbit/s) etc.)
  - Bytes are used to define the size of memory and storage and the rate of parallel communication (bus systems, parallel interfaces).
- Why are 8b (as 1B) so important?
  - The first binary adders have been implemented for 8b.
  - This is why the ASCII table is represented with 8b for convenience, although ASCII is a 7b code (only for the characters needed in the english language). It has 256 characters, and most languages have lesser than 256 characters. The 2nd half of ASCII was used to define non-compatible extensions of non-english languages (e.g. "latin alphabet no. 1"), and some languages still have more than 256 characters (Chinese ideographs). – So a new standard was developed to represent more characters, it is called Unicode and uses multiple planes of 2B each, which makes at least 65.536 characters per plane.
  - Nowadays, we still use 8b-wise grouping of data, but that is just a convention used since the first binary adders! Nobody hindered computing from the beginning using 10, 12 or 4b, e.g. there are systems using 6b!
- Are there other binary systems not only representing numbers?
  - The Morse code (different count of bits per group/letter).
  - The Braille code (six bits are standard (64 combinations), nowadays also eight bits for computers, as more groups/letters are needed (256 combinations)).
  - The DNA's code is defined using two different base pairs, adenine-thymine and guanine-cytosine.

Excursus: Units for Amounts of Data in Computer Systems

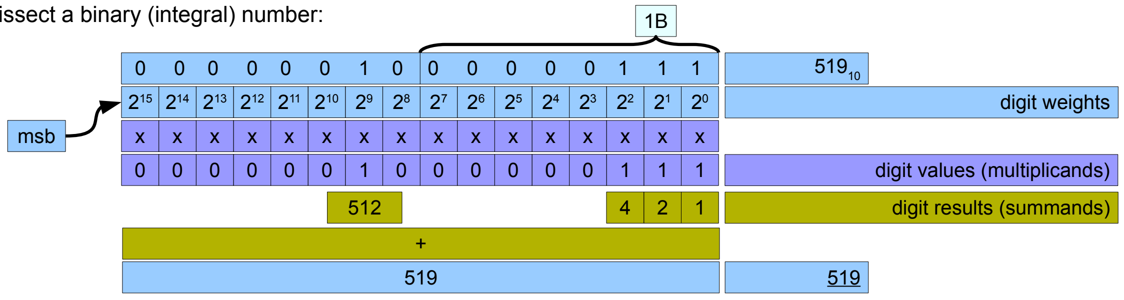
- A single byte does not express a lot of memory, therefor bytes are grouped in orders of magnitude:
  - kilobyte (kB) = 1000B, megabyte (MB) = 1000kB, gigabyte (GB) = 1000GB and so forth
- But, those magnitudes are imprecise, because bytes scale up in magnitudes of base 2, and not of base 10.
  - The magnitudes of 10, i.e. kilo, mega, giga, tera etc., are standardized for SI (Système international d'unités) units only.
- The imprecision shows up, when we recognize, that the SI-prefixes as base 10-scale lessen the actual numbers significantly.
  - A kB makes actually 1024B ($(2^{10})^1$) and not 1000B (10^3B), this is a loss of 2,4%.
- For units of base 2, i.e. the ones to measure data in computer systems, the International Electrotechnical Commission (IEC) prefix system was introduced:

Prefix	Name	Value
Ki	Kibi	1KiB = $(2^{10})^1$ = 1.024B
Mi	Mibi	1MiB = $(2^{10})^2$ =1.048.576B
Gi	Gibi	1GiB = $(2^{10})^3$ = 1.073.741.824B

- So, we have to deal with the units kibibyte (KiB), mibibyte (MiB), gibibyte (GiB) and so forth

The Binary Numeral System

- Let's dissect a binary (integral) number:

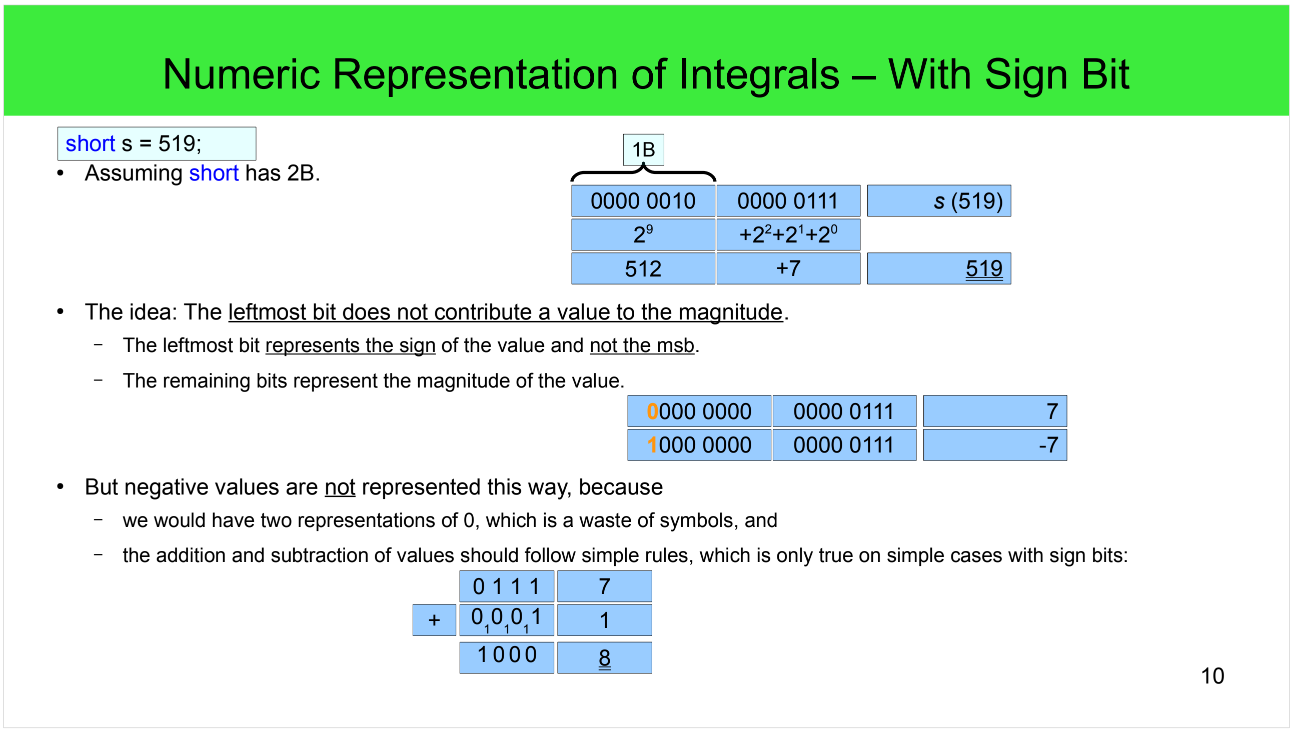


- The binary numeral system has following features:
  - A single binary digit can hold one of the symbols [0, 1].
  - The mathematical notation alternatives of literals look like this: 1000000111_2 or 1000000111_{bin} or 1000000111_{two}.
  - (Here we've a simple encoding of numbers: Binary Coded Decimal digit (BCD).)
  - Binary numbers are read from right to left to avoid misinterpretations!

Good to know
In C++14 integer literals can be written as binary number with the 0b-prefix.

```
int value = 0b1000000111; // value (519)
```

- This is the simple part: the digit weights need only be multiplied by 1 or 0.
  - The msb is called most significant bit, because it contributes the highest-weight digit to the value.
  - The lsb is called least significant bit, because it contributes the smallest-weight digit to the value.



Numeric Representation of Integrals – With Sign Bit

```
short s = 519;
```

- Assuming short has 2B.

1B

0000 0010	0000 0111	*s* (519)
2^9	$+2^2+2^1+2^0$	
512	+7	<u>519</u>

- The idea: The <u>leftmost bit does not contribute a value to the magnitude</u>.
  - The leftmost bit <u>represents the sign</u> of the value and <u>not the msb</u>.
  - The remaining bits represent the magnitude of the value.

0000 0000	0000 0111	7
1000 0000	0000 0111	-7

- But negative values are <u>not</u> represented this way, because
  - we would have two representations of 0, which is a waste of symbols, and
  - the addition and subtraction of values should follow simple rules, which is only true on simple cases with sign bits:

	0 1 1 1	7
+	$0_1 0_1 0_1 1$	1
	1 0 0 0	<u>8</u>

10

- Let's begin by understanding how binary representations look like.
- <u>How is the binary value read?</u>
- <u>As short has 2B, how many combinations are representable?</u>
  - 65536 combinations (the values 0 - 65535)
- Remember that the sizes of fundamental types in C++ are not exactly defined. Info: C99's <stdint.h> defines integer types of guaranteed size, <inttypes.h> defines integer types of specific size and <stdbool.h> defines an explicit boolean type.
- The effective value of a binary representation is the sum of the digits having the base of two to the power of the position in the bit pattern.
- In the notation used above the bits are grouped in four-digit blocks, separating one byte in two halves. A half-byte is often called "nibble" or "nybble" (meaning a small "bite" of something) or "tetrad" or "quadruple" (meaning something consisting of four pieces, i.e. four bits in this case). Nibbles play a special role in representing binary values as hexadecimal values.
  - A set of two bytes (16b) is often called "word" independently of the platform.
- <u>After we know about the sign-bit, how many combinations can short represent?</u>
  - Still 65536 combinations (but the values are -32768 - 32767).
- Binary addition works like decimal addition, we have only to do more carry overs, because we have less digits leading to more carry overs.
- Binary addition is very important as it is basically the only arithmetic operation a CPU can do and is a very simple operation.
  - Virtually, a CPU can only count! – Adding is just the next operation built upon counting.
  - Electronically it can be implemented with XOR-gates for adding and NAND-gates for the carry-overs that make up a half-adder combined with cascaded OR-gates to make it a full-adder. Additions can implement all other arithmetic operations: subtraction → addition of a negative value, multiplication → bit shift and multiple addition, division → bit shift multiple subtraction.
  - If a system could only add, it would only be a calculator. <u>The ability to add and jump to implement loops to repeat operations during clock cycles, makes a calculator a computer.</u>

Numeric Representation of Integrals – One's Complement

- After we discussed how to do bitwise addition, let's inspect bitwise subtraction.
  - Generally the way to subtract values is to add negative values.
  - The problem: bitwise addition does not work with the sign bit:

	0000 0000	0000 0111	7
+	1000 0000	0000 0111	-7
	1000 0000	0000 1110	-14

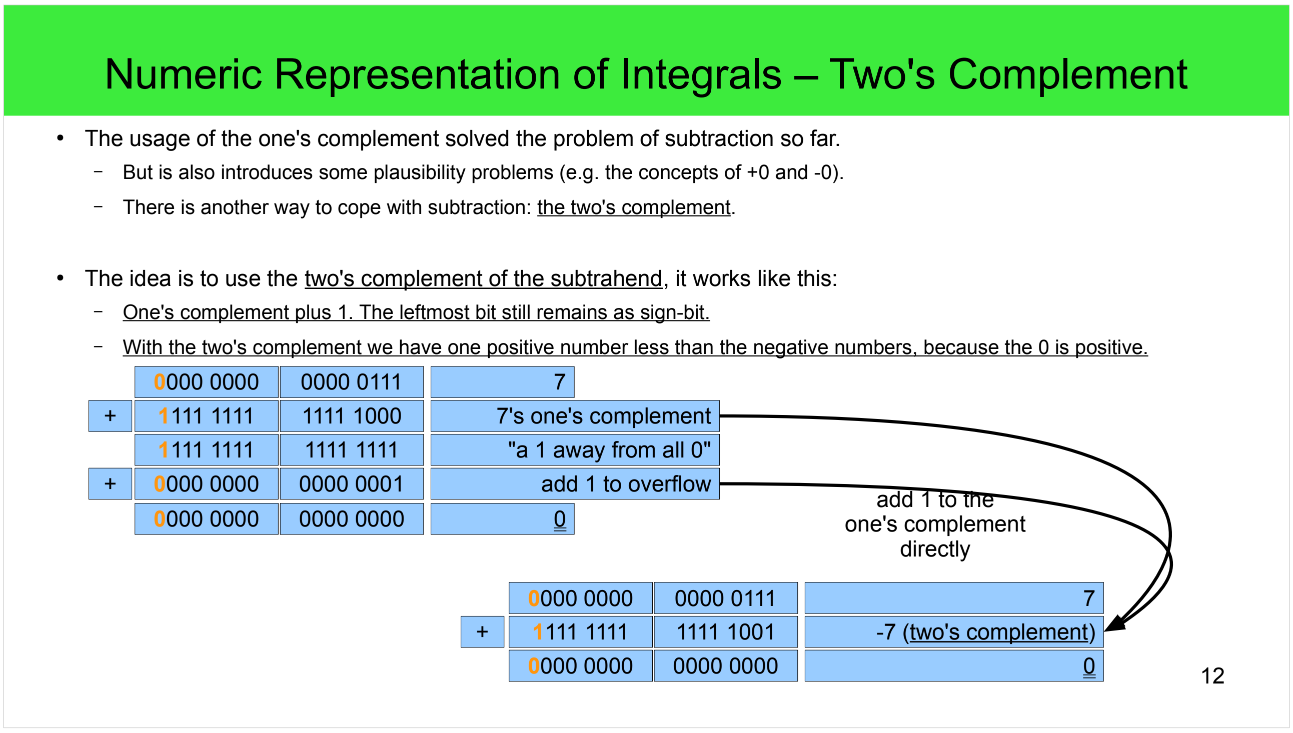
- Another way to represent negative values is the one's complement.
  - The one's complement is just the result of inverting all bits (complement) of a value.

1111 1111	1111 1000	-7

  - The leftmost bit does still represent the sign of the number.

- The next problem: bitwise addition yields another way to represent 0 (all 0 and all 1):

	0000 0000	0000 0111	7
+	1111 1111	1111 1000	-7
	1111 1111	1111 1111	0



Numeric Representation of Integrals – Two's Complement

- The usage of the one's complement solved the problem of subtraction so far.
  - But is also introduces some plausibility problems (e.g. the concepts of +0 and -0).
  - There is another way to cope with subtraction: the two's complement.

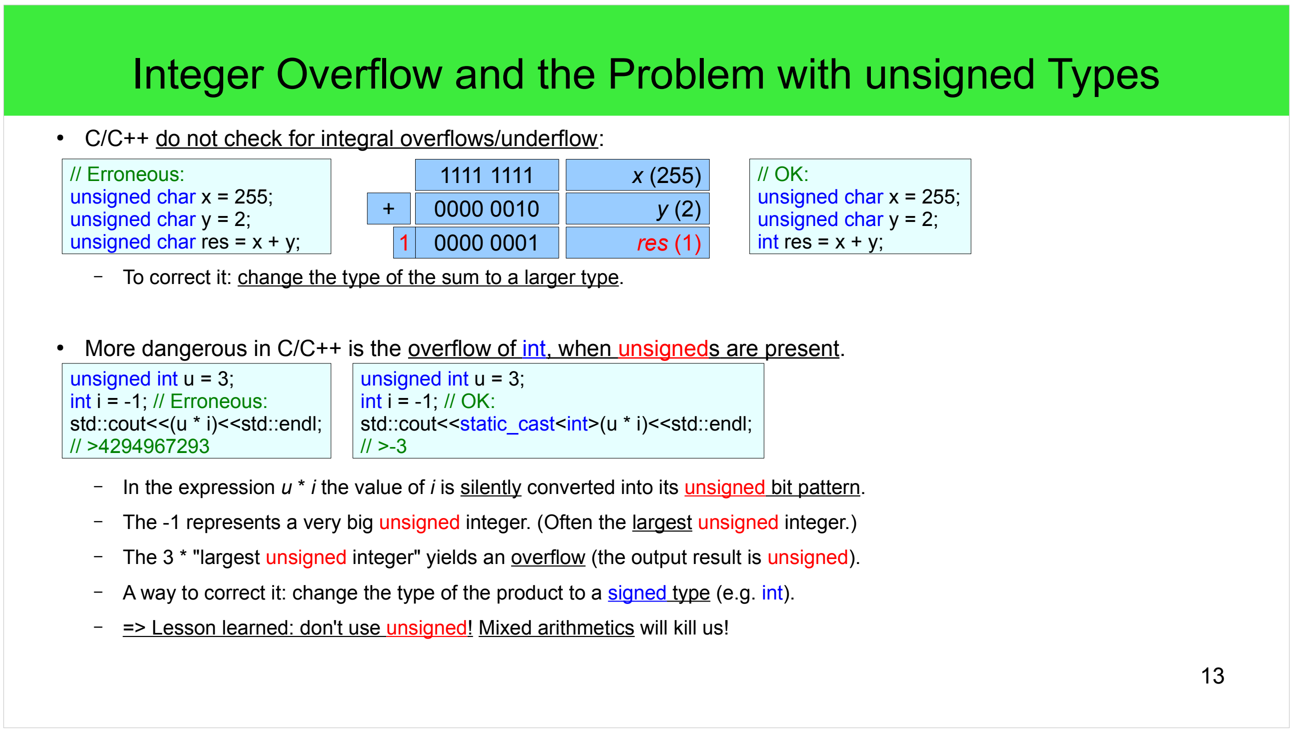
- The idea is to use the two's complement of the subtrahend, it works like this:
  - One's complement plus 1. The leftmost bit still remains as sign-bit.
  - With the two's complement we have one positive number less than the negative numbers, because the 0 is positive.

	0000 0000	0000 0111	7
+	1111 1111	1111 1000	7's one's complement
	1111 1111	1111 1111	"a 1 away from all 0"
+	0000 0000	0000 0001	add 1 to overflow
	0000 0000	0000 0000	0

add 1 to the
one's complement
directly

	0000 0000	0000 0111	7
+	1111 1111	1111 1001	-7 (two's complement)
	0000 0000	0000 0000	0

- How to get the positive seven back from the negative seven presented with the two's complement pattern? -> Invert the pattern and add one.
- The C/C++ type int is usually represented with the two's complement.
- Nowadays the calculation of the two's complement can be directly done in hardware.



Integer Overflow and the Problem with unsigned Types

- C/C++ do not check for integral overflows/underflow:

```
// Erroneous:
unsigned char x = 255;
unsigned char y = 2;
unsigned char res = x + y;
```

	1111 1111	*x* (255)
+	0000 0010	*y* (2)
1	0000 0001	*res* (1)

```
// OK:
unsigned char x = 255;
unsigned char y = 2;
int res = x + y;
```

  - To correct it: change the type of the sum to a larger type.

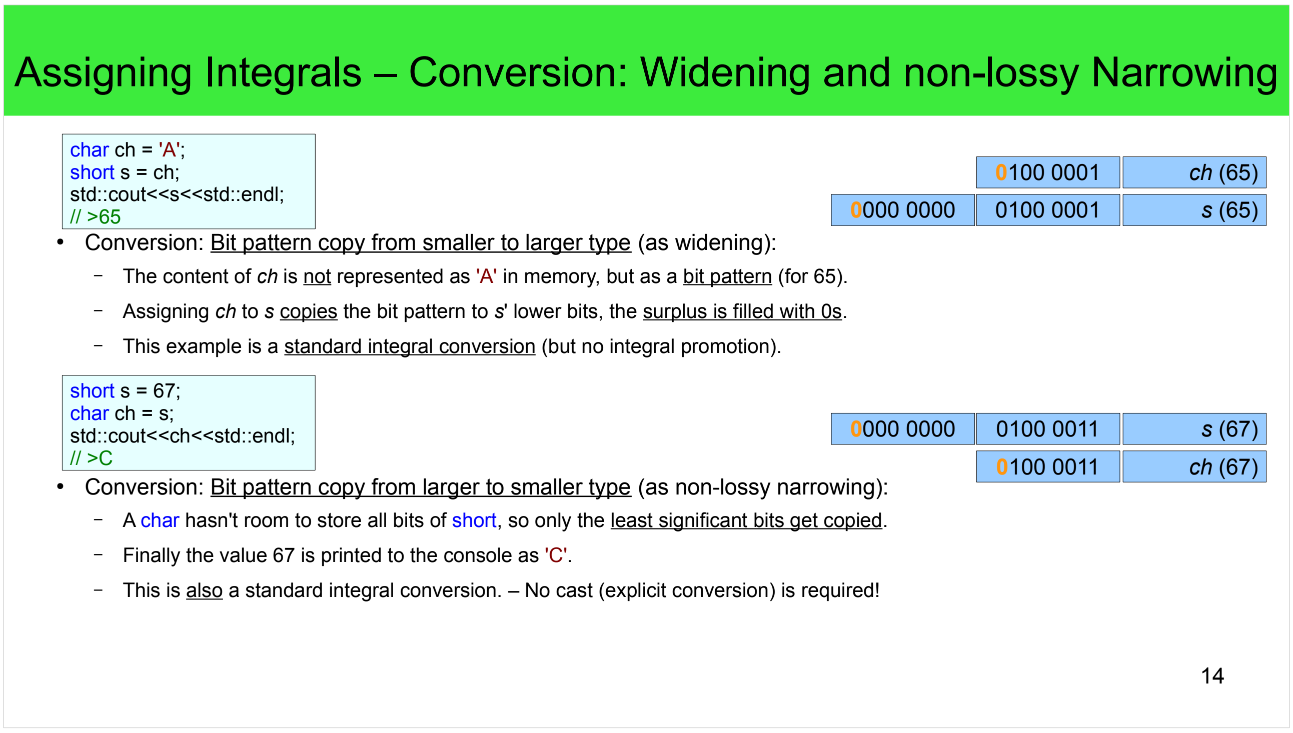
- More dangerous in C/C++ is the overflow of int, when unsigneds are present.

```
unsigned int u = 3;
int i = -1; // Erroneous:
std::cout<<(u * i)<<std::endl;
// >4294967293
```

```
unsigned int u = 3;
int i = -1; // OK:
std::cout<<static_cast<int>(u * i)<<std::endl;
// >-3
```

  - In the expression *u* * *i* the value of *i* is silently converted into its unsigned bit pattern.
  - The -1 represents a very big unsigned integer. (Often the largest unsigned integer.)
  - The 3 * "largest unsigned integer" yields an overflow (the output result is unsigned).
  - A way to correct it: change the type of the product to a signed type (e.g. int).
  - => Lesson learned: don't use unsigned! Mixed arithmetics will kill us!

- The unsigned char example: the assignment of the *res* is ok, because the range of values of unsigned char fits into int (widening).
- Often overflows are errors (but sometimes it is part of the algorithm, e.g. for hash calculation).
- In C/C++ unsigned is often used for hardware near operations.
- In "ordinary" applications, unsigned is sometimes used to gain the ability to store larger values, because one more bit for value representation is available. But if int is not large enough, often unsigned will become too small soon as well. – So, in this case we should design a user defined type in C/C++.
- Bugs introduced with the usage of unsigned are very difficult to spot, just consider the innocent example above. We should always use int as our default integral type!



Assigning Integrals – Conversion: Widening and non-lossy Narrowing

```
char ch = 'A';
short s = ch;
std::cout<<s<<std::endl;
// >65
```

	0100 0001	*ch* (65)
0000 0000	0100 0001	*s* (65)

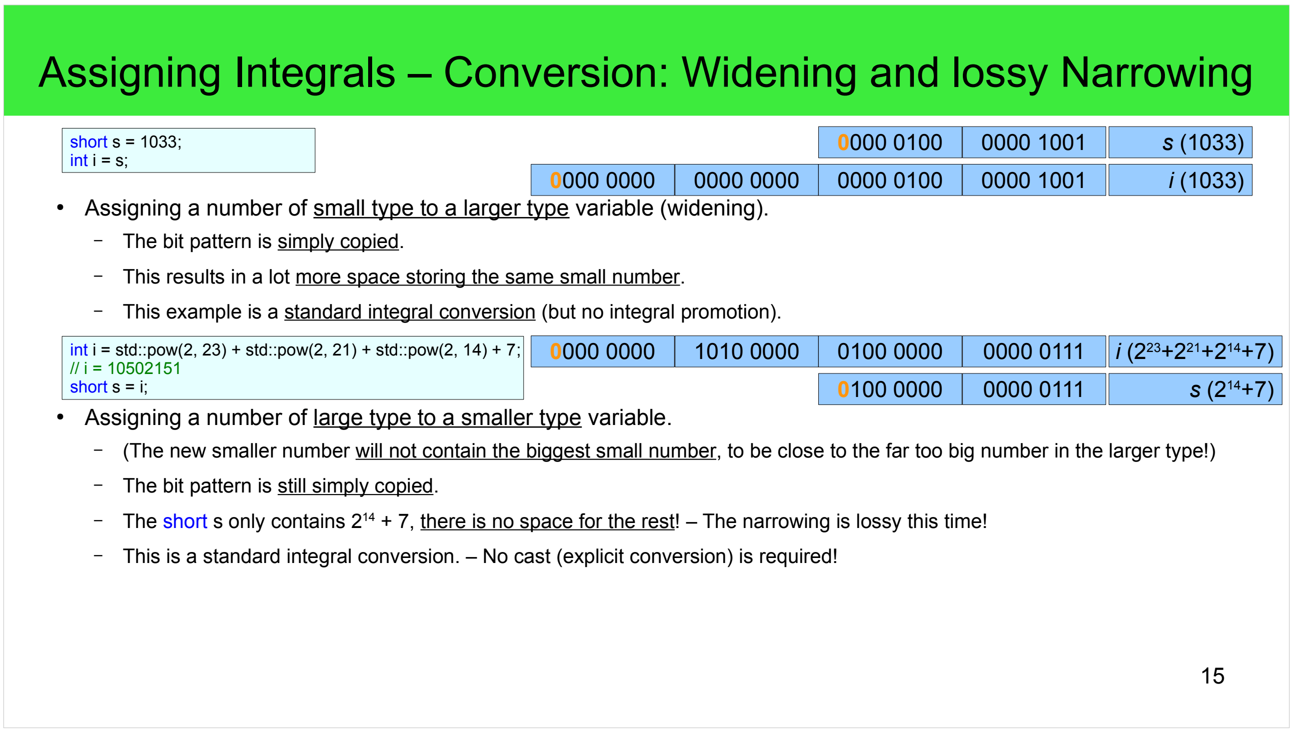
- Conversion: Bit pattern copy from smaller to larger type (as widening):
  - The content of *ch* is not represented as 'A' in memory, but as a bit pattern (for 65).
  - Assigning *ch* to *s* copies the bit pattern to *s*' lower bits, the surplus is filled with 0s.
  - This example is a standard integral conversion (but no integral promotion).

```
short s = 67;
char ch = s;
std::cout<<ch<<std::endl;
// >C
```

0000 0000	0100 0011	*s* (67)
	0100 0011	*ch* (67)

- Conversion: Bit pattern copy from larger to smaller type (as non-lossy narrowing):
  - A char hasn't room to store all bits of short, so only the least significant bits get copied.
  - Finally the value 67 is printed to the console as 'C'.
  - This is also a standard integral conversion. – No cast (explicit conversion) is required!

- In this and the following examples we'll make use of the types char, short and float to understand the effects on bit-patterns. – We already discussed that we should use the fundamental types int and double for everyday programming primarily.
- The char to short example: the assignment is non-lossy, because the range of values of char fits into short (widening).
- C/C++ let us do narrowing conversions without a cast, this makes C/C++-type-languages unsafe! Standard integral conversions can be lossy!
- It should be clarified that in C++ chars and shorts are implicitly promoted to int (or unsigned) in expressions (e.g. binary operations, but also function calls apart from the overload resolution) before the operation takes place. The only exceptions are taking the size (sizeof) and taking the address of char/short.
  - Integral promotions are guaranteed non-lossy, there are only following promotions: from char, signed char, unsigned char, short, unsigned short to int or unsigned int. Esp. int to long is not an integral promotion, but an integral standard conversion (this is the only one that is guaranteed to be non-lossy).
  - int-promotion is performed, because int is the biggest (integral) type that is efficient, and it is guaranteed that no information will be lost.



Assigning Integrals – Conversion: Widening and lossy Narrowing

```
short s = 1033;
int i = s;
```

		0000 0100	0000 1001	*s* (1033)
0000 0000	0000 0000	0000 0100	0000 1001	*i* (1033)

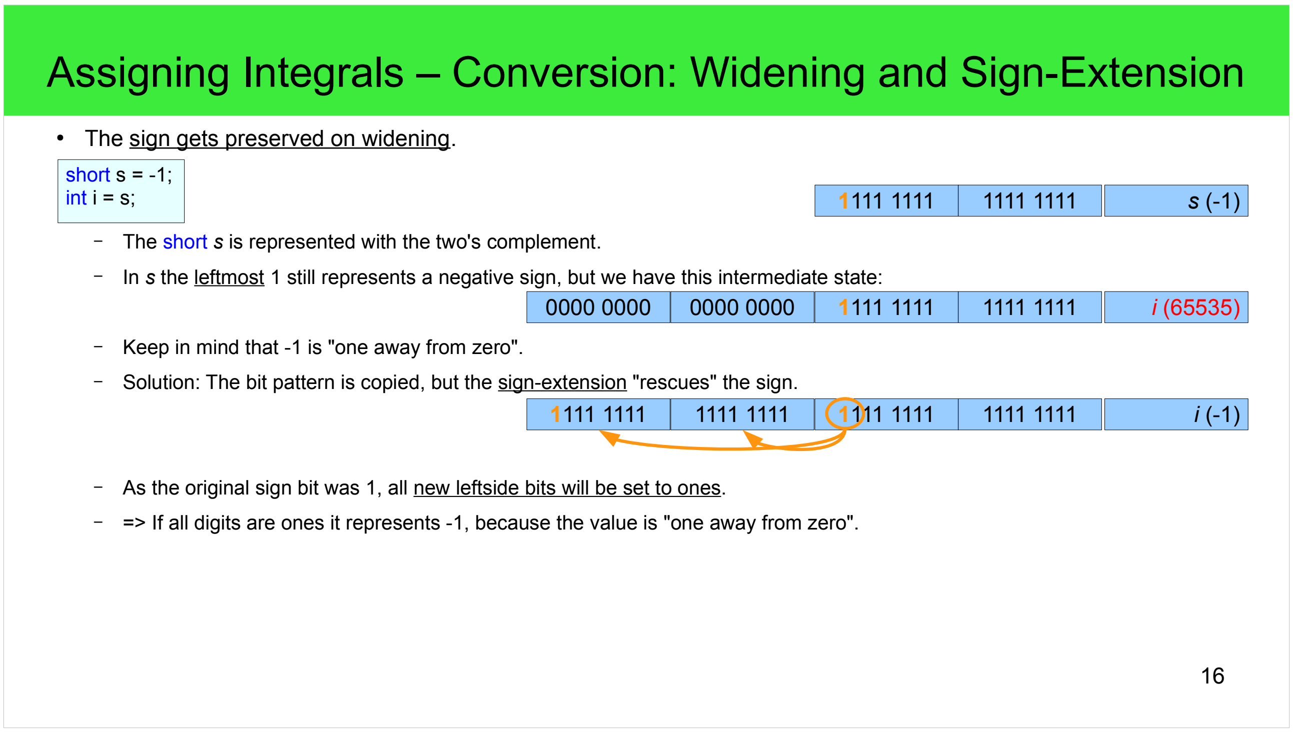
- Assigning a number of small type to a larger type variable (widening).
  - The bit pattern is simply copied.
  - This results in a lot more space storing the same small number.
  - This example is a standard integral conversion (but no integral promotion).

```
int i = std::pow(2, 23) + std::pow(2, 21) + std::pow(2, 14) + 7;
// i = 10502151
short s = i;
```

0000 0000	1010 0000	0100 0000	0000 0111	*i* ($2^{23}+2^{21}+2^{14}+7$)
		0100 0000	0000 0111	*s* ($2^{14}+7$)

- Assigning a number of large type to a smaller type variable.
  - (The new smaller number will not contain the biggest small number, to be close to the far too big number in the larger type!)
  - The bit pattern is still simply copied.
  - The short s only contains 2^{14} + 7, there is no space for the rest! – The narrowing is lossy this time!
  - This is a standard integral conversion. – No cast (explicit conversion) is required!

- In other type-safe languages (e.g. Java or C#) we had to cast each narrowing conversion (i.e. we had to cast the int value to a float value).



Assigning Integrals – Conversion: Widening and Sign-Extension

- The sign gets preserved on widening.

```
short s = -1;
int i = s;
```

1111 1111	1111 1111	*s* (-1)

  - The short *s* is represented with the two's complement.
  - In *s* the leftmost 1 still represents a negative sign, but we have this intermediate state:

0000 0000	0000 0000	1111 1111	1111 1111	*i* (65535)

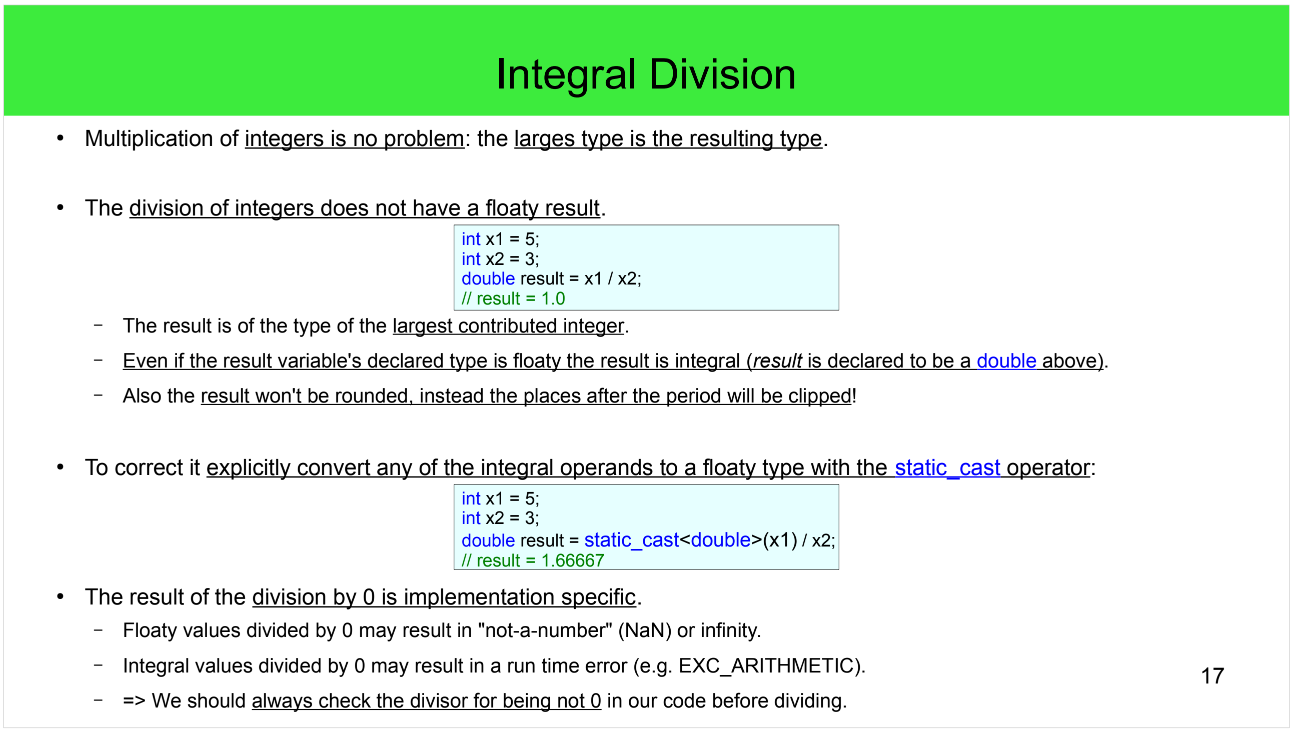
  - Keep in mind that -1 is "one away from zero".
  - Solution: The bit pattern is copied, but the sign-extension "rescues" the sign.

1111 1111	1111 1111	1111 1111	1111 1111	*i* (-1)

  - As the original sign bit was 1, all new leftside bits will be set to ones.
  - => If all digits are ones it represents -1, because the value is "one away from zero".

16

- Sign extension also happens with positive numbers, but the extension with 0's to the new "leftside" bits is more obvious.



Integral Division

- Multiplication of integers is no problem: the larges type is the resulting type.
- The division of integers does not have a floaty result.

```
int x1 = 5;
int x2 = 3;
double result = x1 / x2;
// result = 1.0
```

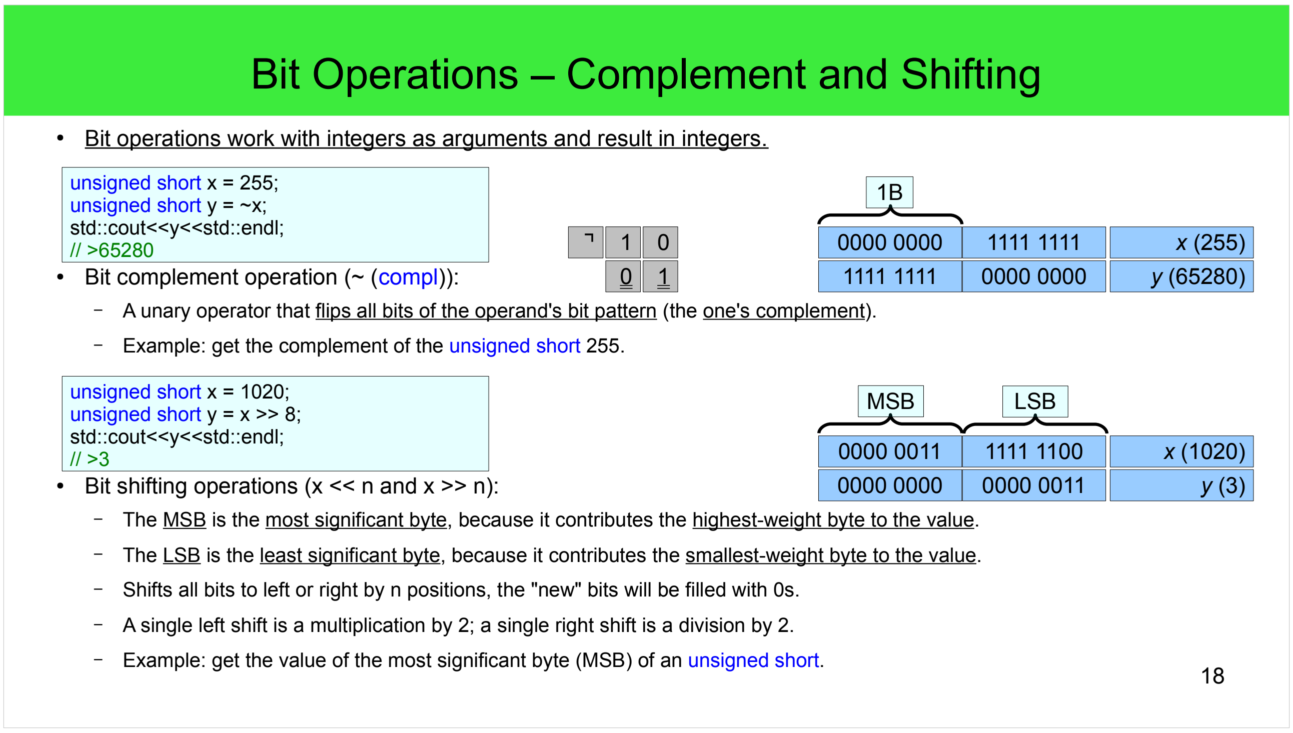
  - The result is of the type of the largest contributed integer.
  - Even if the result variable's declared type is floaty the result is integral (*result* is declared to be a double above).
  - Also the result won't be rounded, instead the places after the period will be clipped!
- To correct it explicitly convert any of the integral operands to a floaty type with the static_cast operator:

```
int x1 = 5;
int x2 = 3;
double result = static_cast<double>(x1) / x2;
// result = 1.66667
```

- The result of the division by 0 is implementation specific.
  - Floaty values divided by 0 may result in "not-a-number" (NaN) or infinity.
  - Integral values divided by 0 may result in a run time error (e.g. EXC_ARITHMETIC).
  - => We should always check the divisor for being not 0 in our code before dividing.

17

- The integral result of the integral division makes sense, because how should the integral division know something about floaty types?
- Clipping: the result of 5 : 2 is $1.\overline{6}$, but the part after the period will be clipped away.
- The division by 0 is not "not allowed" in maths, instead it is "just" undefined.



Bit Operations – Complement and Shifting

- Bit operations work with integers as arguments and result in integers.

```
unsigned short x = 255;
unsigned short y = ~x;
std::cout<<y<<std::endl;
// >65280
```

¬	1	0
	0	1

1B		
0000 0000	1111 1111	*x* (255)
1111 1111	0000 0000	*y* (65280)

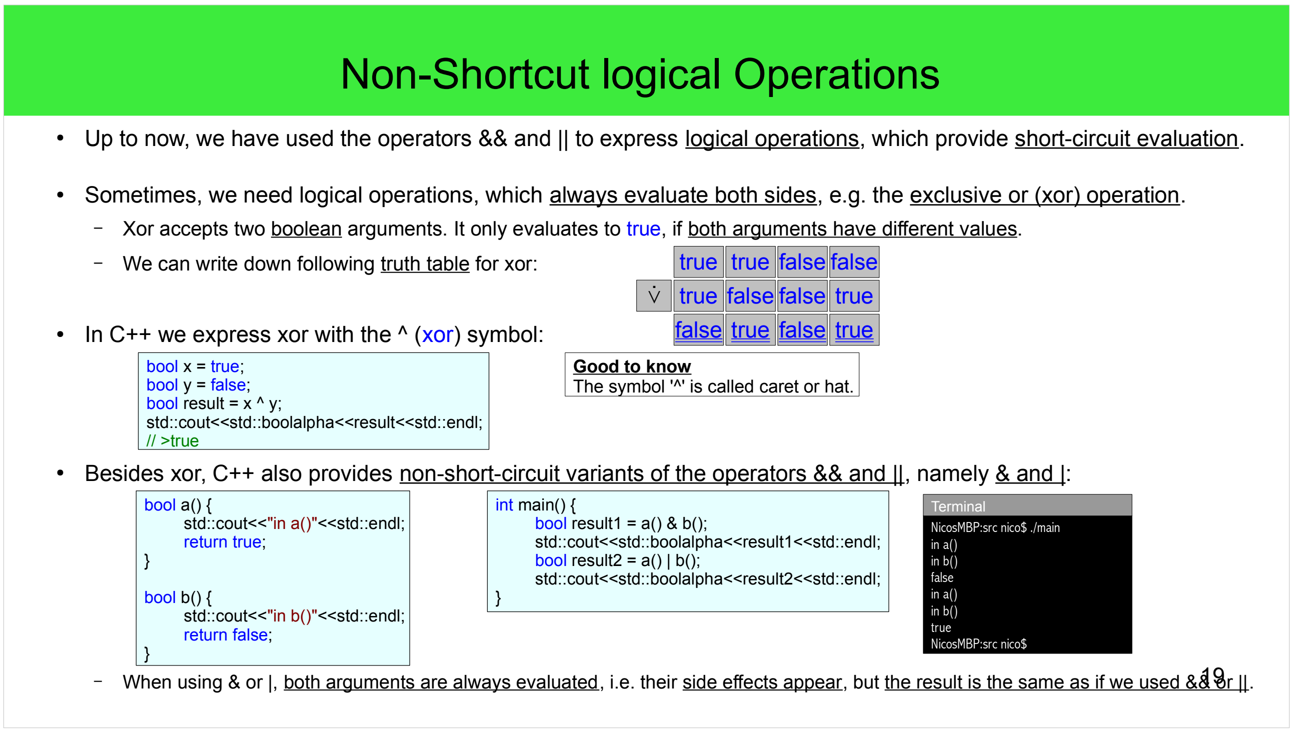
- Bit complement operation (~ (compl)):
  - A unary operator that flips all bits of the operand's bit pattern (the one's complement).
  - Example: get the complement of the unsigned short 255.

```
unsigned short x = 1020;
unsigned short y = x >> 8;
std::cout<<y<<std::endl;
// >3
```

MSB	LSB	
0000 0011	1111 1100	*x* (1020)
0000 0000	0000 0011	*y* (3)

- Bit shifting operations (x << n and x >> n):
  - The MSB is the most significant byte, because it contributes the highest-weight byte to the value.
  - The LSB is the least significant byte, because it contributes the smallest-weight byte to the value.
  - Shifts all bits to left or right by n positions, the "new" bits will be filled with 0s.
  - A single left shift is a multiplication by 2; a single right shift is a division by 2.
  - Example: get the value of the most significant byte (MSB) of an unsigned short.

- In the upcoming example we'll use objects of type unsigned short or unsigned char as so called "bit fields".
- Why do we use unsigned short in these examples?
  - Because we don't want to struggle with the sign bits here.
  - The result of bit operations does also depend on the size and sign of the used values, e.g. the complement of an int value of 255 is different from the complement of an unsigned short value of 255!
- Bit operations are very costly for the CPU as they need many CPU cycles to be processed. As C/C++ allow this kind of operations (also with pointers) in order to mimic CPU operations very closely, they're often called "high-level assembly languages".
- Bit shifts do not rotate the bit pattern, i.e. they are lossy.
- In C/C++ the result of shifting a signed integer is implementation specific. The sign bit could be shifted or preserved. In Java bit shifts preserve the sign, the special bit shift operator >>> shifts the sign (its called "signed right shift").



Non-Shortcut logical Operations

- Up to now, we have used the operators && and || to express logical operations, which provide short-circuit evaluation.
- Sometimes, we need logical operations, which always evaluate both sides, e.g. the exclusive or (xor) operation.
  - Xor accepts two boolean arguments. It only evaluates to true, if both arguments have different values.
  - We can write down following truth table for xor:

	true	true	false	false
$\dot{\vee}$	true	false	false	true
	false	true	false	true

- In C++ we express xor with the ^ (xor) symbol:

```
bool x = true;
bool y = false;
bool result = x ^ y;
std::cout<<std::boolalpha<<result<<std::endl;
// >true
```

Good to know
The symbol '^' is called caret or hat.

- Besides xor, C++ also provides non-short-circuit variants of the operators && and ||, namely & and |:

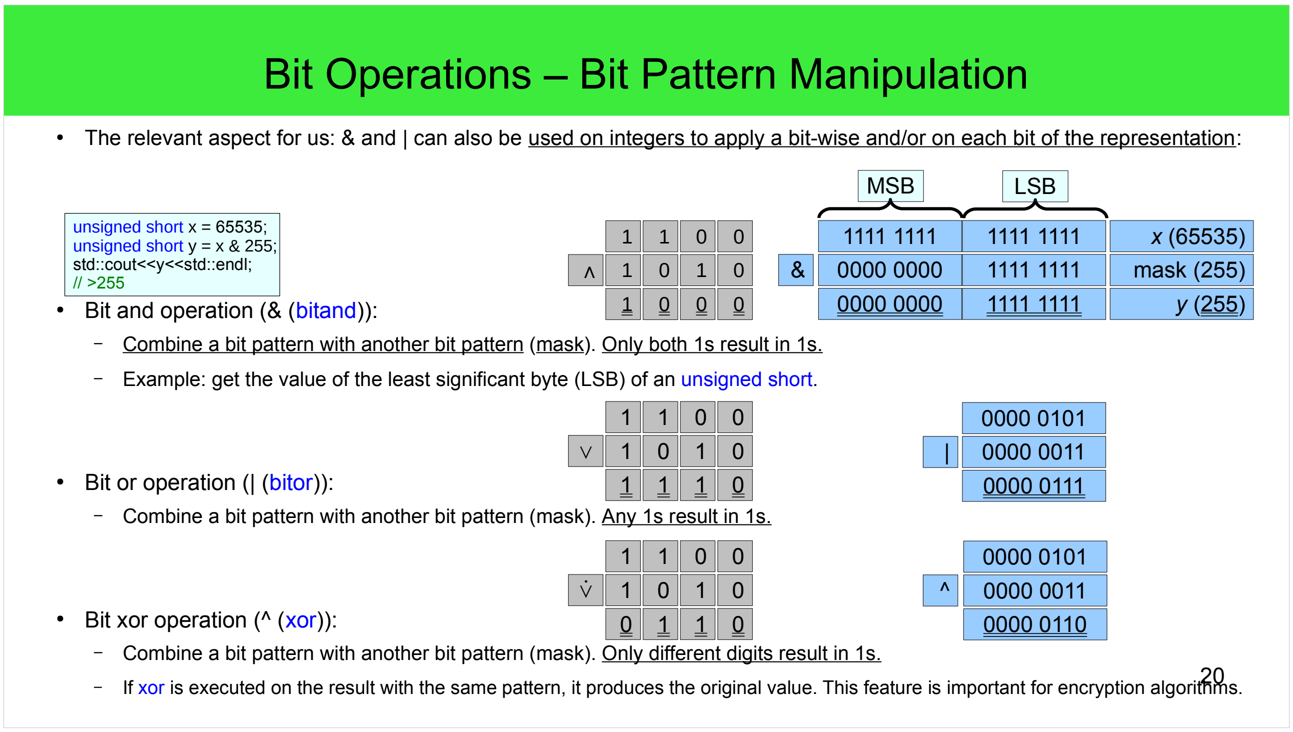
```
bool a() {
	std::cout<<"in a()"<<std::endl;
	return true;
}

bool b() {
	std::cout<<"in b()"<<std::endl;
	return false;
}
```

```
int main() {
	bool result1 = a() & b();
	std::cout<<std::boolalpha<<result1<<std::endl;
	bool result2 = a() | b();
	std::cout<<std::boolalpha<<result2<<std::endl;
}
```

Terminal
```
NicosMBP:src nico$ ./main
in a()
in b()
false
in a()
in b()
true
NicosMBP:src nico$
```

  - When using & or |, both arguments are always evaluated, i.e. their side effects appear, but the result is the same as if we used && or ||.

- There exist different notations of the xor operator in mathematics.



Bit Operations – Bit Pattern Manipulation

- The relevant aspect for us: & and | can also be used on integers to apply a bit-wise and/or on each bit of the representation:

```
unsigned short x = 65535;
unsigned short y = x & 255;
std::cout<<y<<std::endl;
// >255
```

- Bit and operation (& (bitand)):
  - Combine a bit pattern with another bit pattern (mask). Only both 1s result in 1s.
  - Example: get the value of the least significant byte (LSB) of an unsigned short.

	1	1	0	0
∧	1	0	1	0
	1	0	0	0

	MSB	LSB	
	1111 1111	1111 1111	*x* (65535)
&	0000 0000	1111 1111	mask (255)
	0000 0000	1111 1111	*y* (255)

- Bit or operation (| (bitor)):
  - Combine a bit pattern with another bit pattern (mask). Any 1s result in 1s.

	1	1	0	0
∨	1	0	1	0
	1	1	1	0

	0000 0101
|	0000 0011
	0000 0111

- Bit xor operation (^ (xor)):
  - Combine a bit pattern with another bit pattern (mask). Only different digits result in 1s.
  - If xor is executed on the result with the same pattern, it produces the original value. This feature is important for encryption algorithms.

	1	1	0	0
∨̇	1	0	1	0
	0	1	1	0

	0000 0101
^	0000 0011
	0000 0110

20

- Think: a bit-wise operator just applies the respective logical operation on each bit of the values to be processed (assuming 1 -> true, 0 -> false).
- Keep in mind that the bitwise & and | have no short circuit evaluation. => Avoid using them for logical operations (however, there are exceptions from this rule).
- Xor is a very important operation for encryption. ICs for encryption use special circuits for xor operations in the hardware. – People thought, this is a secure way to hide encryption, because the hardware is much more covered than software, which could be read from outside. – However, when one grinds the layers of such an IC, it is simple to identify the xor circuits and reverse engineer the encryption algorithm.

Bit Operations – Bit Flags – Representation and Setting

- We can represent data as a set of on/off switches.
  - This can be expressed as bit pattern whereby each bit represents a switch.

```
unsigned char options = 0;
```

- In a byte (unsigned char in this example) we can have 256 switch combinations.
  - Each single option is represented as a one-bit pattern.
  - In C/C++ options are traditionally represented with enum constants.
  - This bit pattern can be evaluated and manipulated with bit operations.

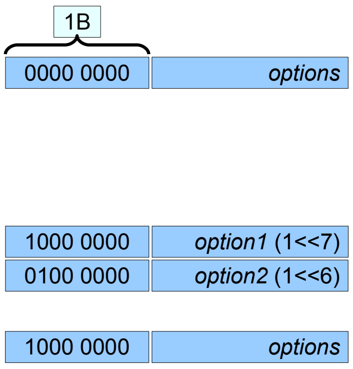
```
unsigned char option1 = 1 << 7;
```

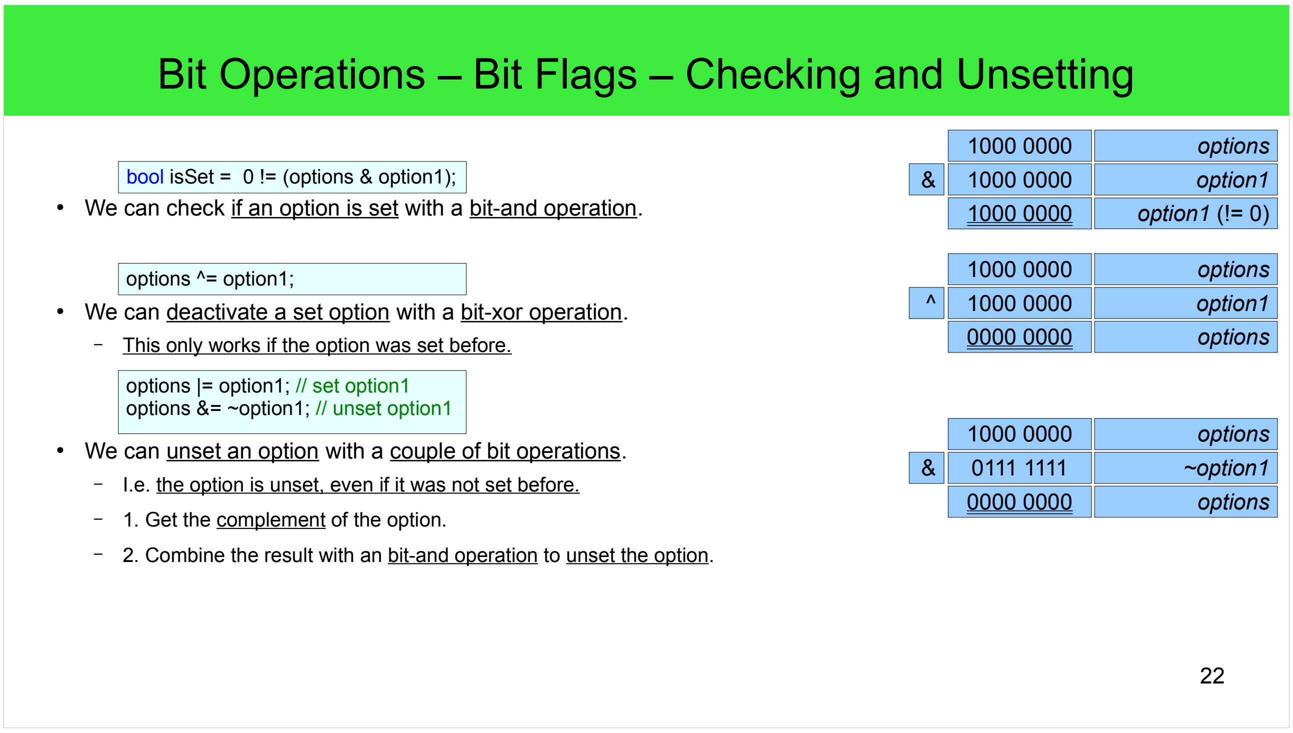
```
unsigned char option2 = 1 << 6;
```

- The representation of bit patterns can be created with shift operations.

```
options |= option1;
```

- We can set an option with a bit-or operation.





Bit Operations – Bit Flags – Checking and Unsetting

```
bool isSet =  0 != (options & option1);
```

- We can check if an option is set with a bit-and operation.

	1000 0000	*options*
&	1000 0000	*option1*
	1000 0000	*option1* (!= 0)

```
options ^= option1;
```

- We can deactivate a set option with a bit-xor operation.
  - This only works if the option was set before.

	1000 0000	*options*
^	1000 0000	*option1*
	0000 0000	*options*

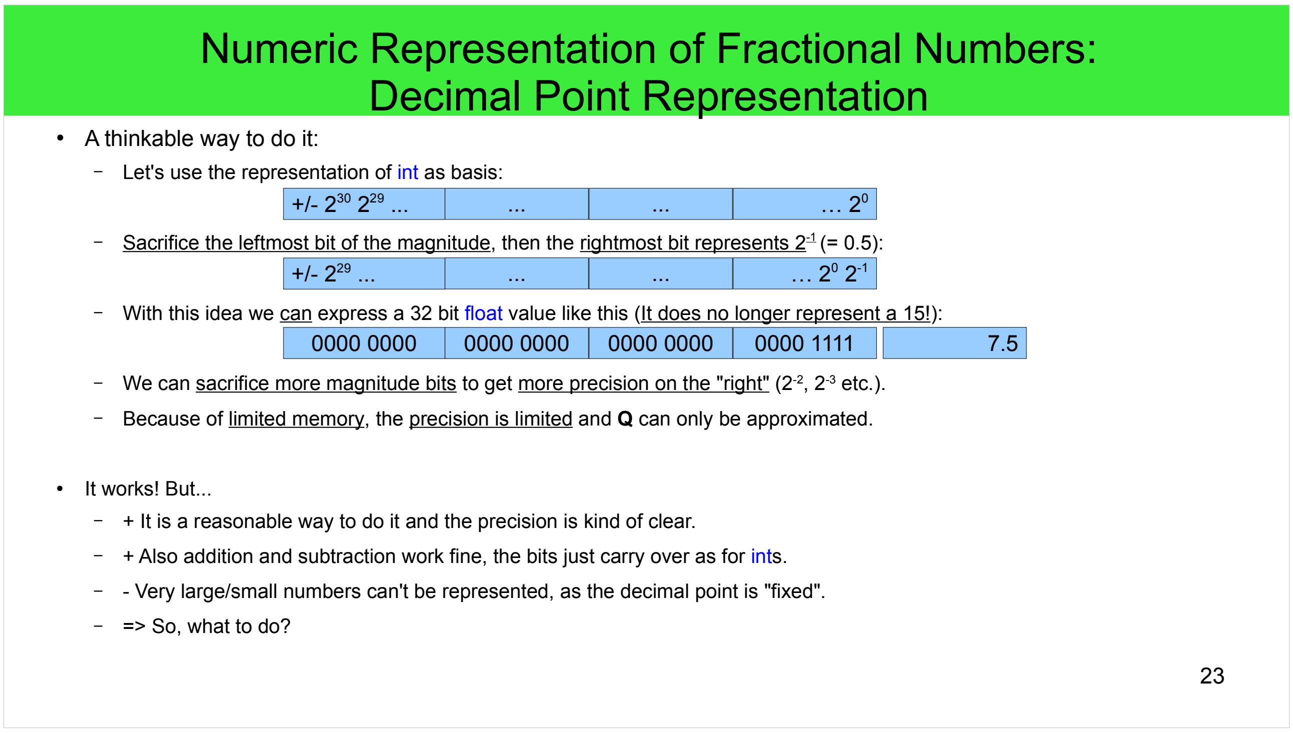
```
options |= option1; // set option1
options &= ~option1; // unset option1
```

- We can unset an option with a couple of bit operations.
  - I.e. the option is unset, even if it was not set before.
  - 1. Get the complement of the option.
  - 2. Combine the result with an bit-and operation to unset the option.

	1000 0000	*options*
&	0111 1111	*~option1*
	0000 0000	*options*

22

- The usage of bit fields this way is indeed interesting, but not very useful in practice:
  - Bit operations are expensive! The compacted memory representation comes for the prize of costly machine code.
  - Bit fields can have individual options that are logically mutually exclusive.
  - Bit fields are not extendable. If n-options of an n bit field are already used, we can not add more options.
  - => In the end better concepts are present in C++, esp. the encapsulation of options with user defined types.
- In the STL these kinds bit fields are used to represent options of streams.



Numeric Representation of Fractional Numbers: Decimal Point Representation

- A thinkable way to do it:
  - Let's use the representation of int as basis:

+/- 2^{30} 2^{29} ...	...	...	... 2^{0}

  - Sacrifice the leftmost bit of the magnitude, then the rightmost bit represents 2^{-1} (= 0.5):

+/- 2^{29} ...	...	...	... 2^{0} 2^{-1}

  - With this idea we can express a 32 bit float value like this (It does no longer represent a 15!):

0000 0000	0000 0000	0000 0000	0000 1111	7.5

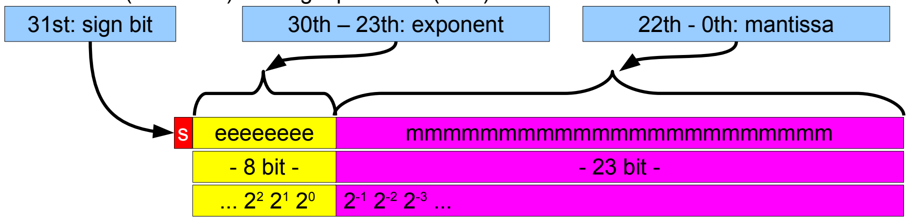
  - We can sacrifice more magnitude bits to get more precision on the "right" (2^{-2}, 2^{-3} etc.).
  - Because of limited memory, the precision is limited and **Q** can only be approximated.
- It works! But...
  - + It is a reasonable way to do it and the precision is kind of clear.
  - + Also addition and subtraction work fine, the bits just carry over as for ints.
  - - Very large/small numbers can't be represented, as the decimal point is "fixed".
  - => So, what to do?

23

- Approximating **Q**: we could, e.g., store π only with a limited precision.
- Decimal point numbers are not that bad. As they can represent numbers with a guaranteed precision, they are often used in problem domains where "stable precision" is urgently required, e.g. financial affairs. – The programming language COBOL, which was esp. designed for financial and business affairs, uses fixed point arithmetics. In C# the datatype decimal (instead of double) implements a decimal point representation. In Java the UDT *BigDecimal* implements a decimal point representation.
- Fixed point type definition in the programming language Ada: type MY_FIXED is delta 0.1 range -40.0..120.0;

Numeric Representation of Fractional Numbers: Binary Floating Point Representation

- IEEE standardized this (IEEE 754) for single precision (float) like so:



- The exponent represents the magnitude of the number.
  - The exponent is an eight bit unsigned integer.
  - It has the range]0, 255[(0 and 255 are reserved), it is interpreted as]-126, 127[. This 127-bias of the exponent allows representing very small and very large numbers.

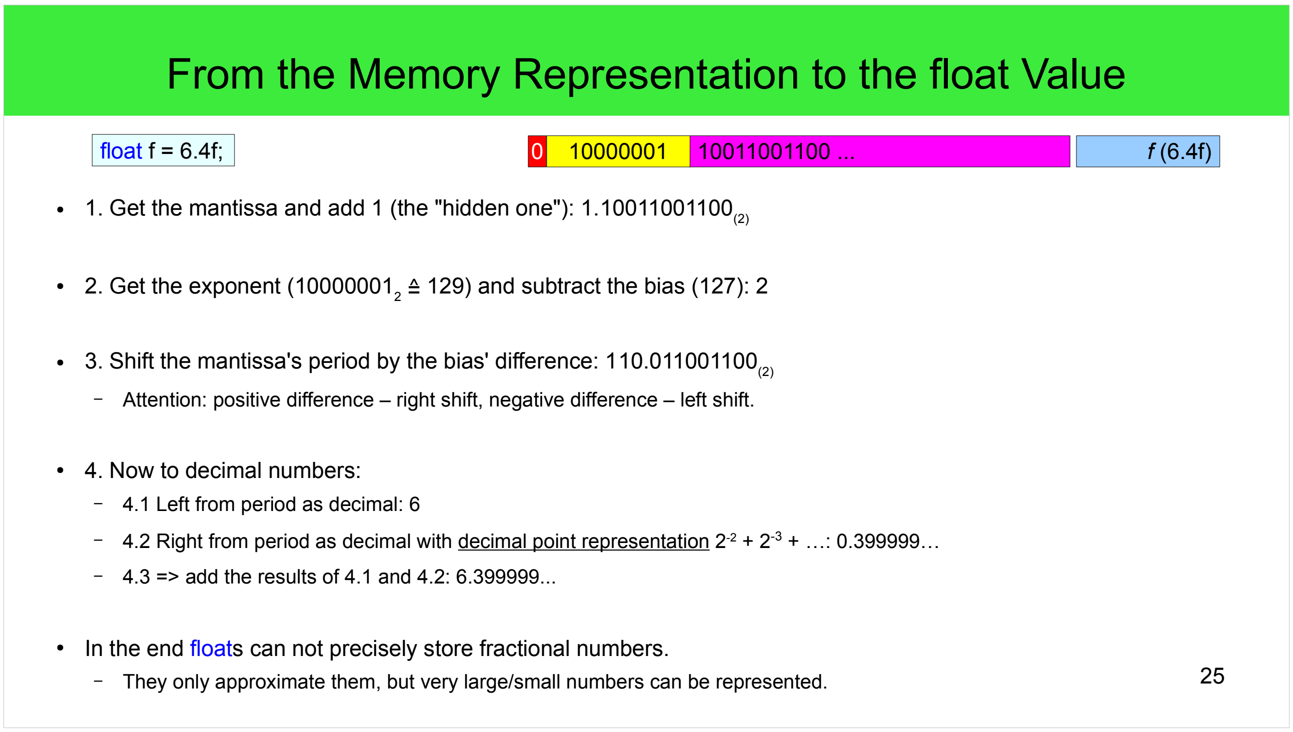
$$\boxed{bias = 2^{\#e-1} - 1 = 2^{8-1} - 1 = \underline{127}}$$

- The mantissa represents the fractional part of the multiplicand.
  - It represents a fractional number from 0.0 to $0.\overline{9}$ as best as possible with 23 bit.

- The represented value is calculated like so:

$$\boxed{(-1)^s \cdot (1_{hidden} + mantissa) \cdot 2^{exponent\ -\ bias}}$$

- Today IEEE 754 (Institute of Electrical and Electronics Engineers) is de facto the industry standard to represent floating point numbers and to define how arithmetics work with floating point numbers.
- The IEEE 754 for binary floating point numbers really uses a sign bit and no other arcane representation.
- The fractional part does not "directly" contribute to the fractional number. It is completely different from the representation of decimal point numbers! The fractional part is only a multiplicand to implement a floating point representation together with the exponent. See the formula!
- Sometimes the exponent is called magnitude.
- With this representation a 32b float covers the range of values from 1.5E-45 to 3.4E38 at a precision of 6 - 8 digits.
- The mantissa only represents the fractional part of the multiplicand w/o the 1 left from the period (this 1 is not encoded, often called the "hidden one"). This is called "normalized representation".
- The signed and biased exponent is also called characteristic or scale. Some words on that:
  - Here the bias is used instead of the two's complement. The usage of a bias is implementation specific.
  - For the exponent (exp) -127 and 128 (0 and 255) are reserved. These represent: exp = 0 and mantissa = 0: the value is 0.0, depending on the sign we have a +0.0 and a -0.0; exp = 255 and mantissa = 0: +/- infinity, depending on the sign bit; exp = 0 and mantissa != 0: the number is valid, but not normalized; exp = 255 and mantissa != 0: the result is NaN (NaN is the the result of invalid operations, as the square root of a negative value, the logarithm of a negative number, things like infinity minus infinity or .0 divided by .0, btw. other numbers divided by .0 result in the value infinity)
  - This range is due to the fact that the full exponent of 2 in the formula could be very large (127) or very small (-126). – It scales the resulting number being very large or very small.



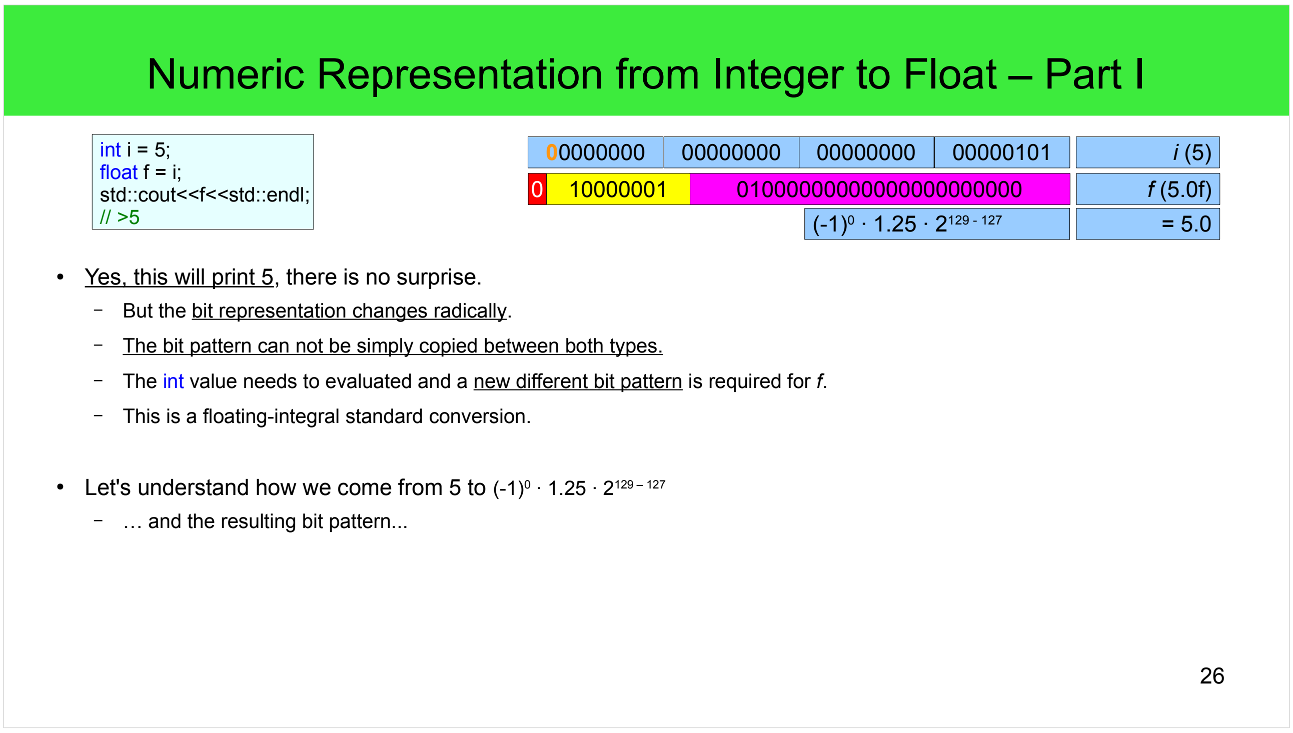
From the Memory Representation to the float Value

`float f = 6.4f;`

0	10000001	10011001100 ...	*f* (6.4f)

- 1. Get the mantissa and add 1 (the "hidden one"): $1.10011001100_{(2)}$
- 2. Get the exponent ($10000001_2 \triangleq 129$) and subtract the bias (127): 2
- 3. Shift the mantissa's period by the bias' difference: $110.011001100_{(2)}$
  - Attention: positive difference – right shift, negative difference – left shift.
- 4. Now to decimal numbers:
  - 4.1 Left from period as decimal: 6
  - 4.2 Right from period as decimal with decimal point representation $2^{-2} + 2^{-3} + \dots$: 0.399999…
  - 4.3 => add the results of 4.1 and 4.2: 6.399999...
- In the end floats can not precisely store fractional numbers.
  - They only approximate them, but very large/small numbers can be represented.

- The ellipsis right from the mantissa is not significant.
- The precision of float is only six digits, so 6.399999 is a float value representing 6.4f as 6.399999 has seven digits. Keep in mind that all digits (also those left from the period) of the number contribute to the precision-significant digits. If we have more than the precision-significant digits in a float result, it will be typically represented in scientific notation, getting more and more imprecise.
  - E.g. the x87 FPUs do internally calculate with an 80b format in order to minimize cumulative rounding errors as a result of iterative float operations.
- In the end the benefits of the binary floating point representation (very large/small numbers) are the downsides of the decimal point representation (benefit: controlled precision) and vice versa.



Numeric Representation from Integer to Float – Part I

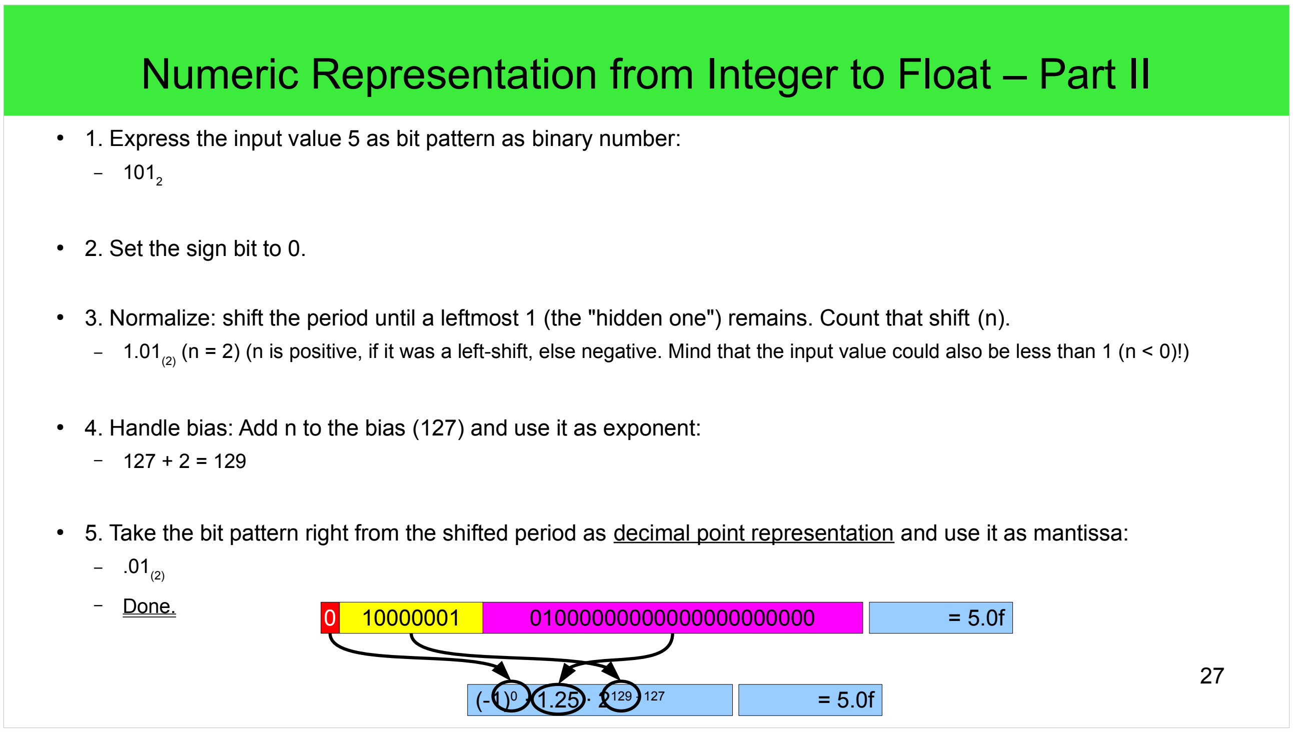
```
int i = 5;
float f = i;
std::cout<<f<<std::endl;
// >5
```

00000000	00000000	00000000	00000101	*i* (5)
0	10000001	01000000000000000000000		*f* (5.0f)
		$(-1)^0 \cdot 1.25 \cdot 2^{129 - 127}$		= 5.0

- Yes, this will print 5, there is no surprise.
  - But the bit representation changes radically.
  - The bit pattern can not be simply copied between both types.
  - The int value needs to evaluated and a new different bit pattern is required for *f*.
  - This is a floating-integral standard conversion.
- Let's understand how we come from 5 to $(-1)^0 \cdot 1.25 \cdot 2^{129 – 127}$
  - … and the resulting bit pattern...

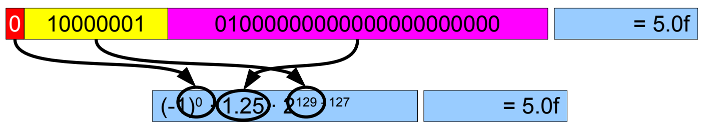
26

- We can configure the display of digits after the period with the manipulators *std::fixed* together with *std::setprecision()*: *std::cout<<std::fixed<<std::setprecision(2)<<f<<std::endl*;
- Note that conversions between floaty and integral types are standard conversions and no promotions! – But a float can be promoted to a double.

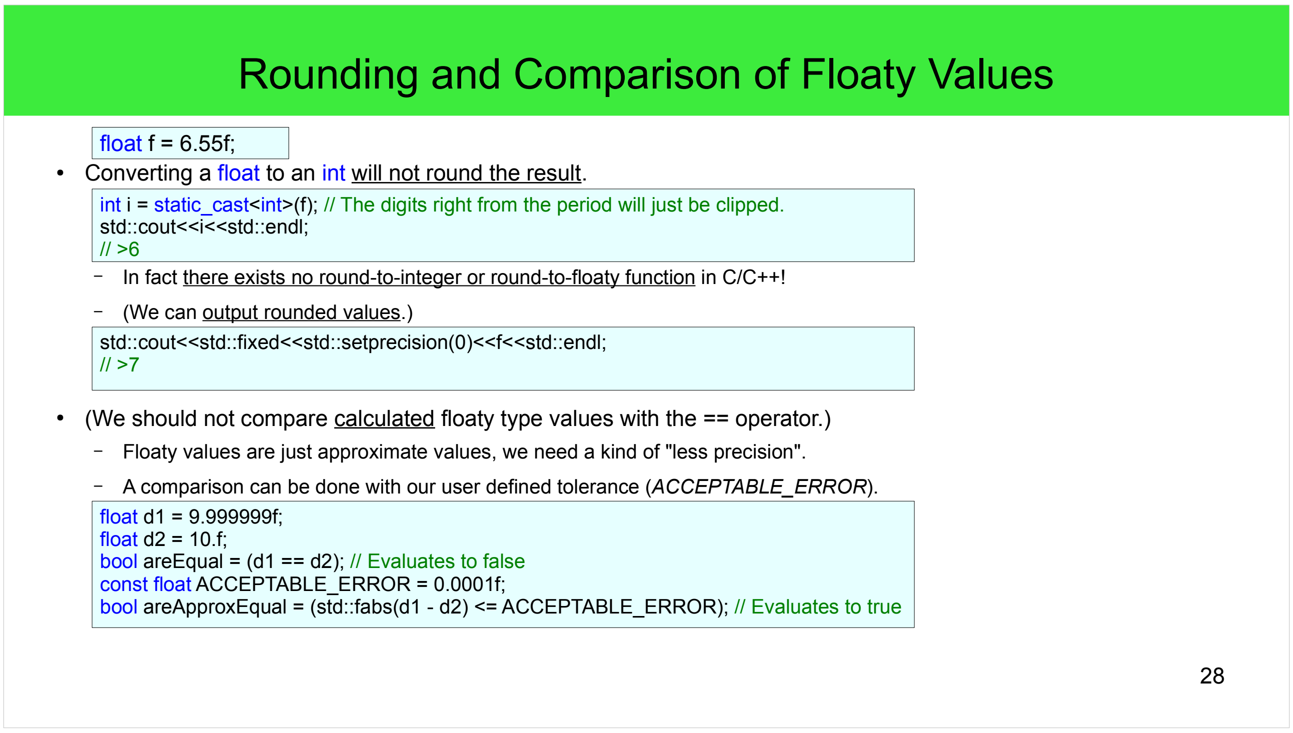


Numeric Representation from Integer to Float – Part II

- 1. Express the input value 5 as bit pattern as binary number:
  - 101_2
- 2. Set the sign bit to 0.
- 3. Normalize: shift the period until a leftmost 1 (the "hidden one") remains. Count that shift (n).
  - $1.01_{(2)}$ (n = 2) (n is positive, if it was a left-shift, else negative. Mind that the input value could also be less than 1 (n < 0)!)
- 4. Handle bias: Add n to the bias (127) and use it as exponent:
  - 127 + 2 = 129
- 5. Take the bit pattern right from the shifted period as decimal point representation and use it as mantissa:
  - $.01_{(2)}$
  - Done.



- In the final calculation there is a 1.25. – What's the source of the 1.? – Well, this is the "hidden one", which is just required to make the calculation work. Its presence is usually just implied.
- In this example we can see why the IEEE 754 presentation is useful: only the comparison of the sign and the exponent of different numbers make a comparison very simple. - The count of digits is represented there, if these counts are equal the mantissas can be compared.
- Nevertheless these floaty operations are more complicated then the equivalent integer operations. In past these operations have been coded as little machine language programs with integer arithmetics (e.g. as software). Today an extra part of the CPU, the "Floating-point unit" (FPU), deals with floaty operations completely in hardware, which is much faster than the software-way!
  - First FPUs where extern co-processors having an own socket near the CPU (Intel: x87 FPUs for x86 CPUs and Motorola: 68881/68882 FPUs for 68000 CPUs). Later, FPUs have been integrated into the CPU (since Intel 80486DX (the internal FPU was said to be x87 compatible) and since Motorola 68040).



Rounding and Comparison of Floaty Values

```
float f = 6.55f;
```

- Converting a float to an int <u>will not round the result</u>.

```
int i = static_cast<int>(f); // The digits right from the period will just be clipped.
std::cout<<i<<std::endl;
// >6
```

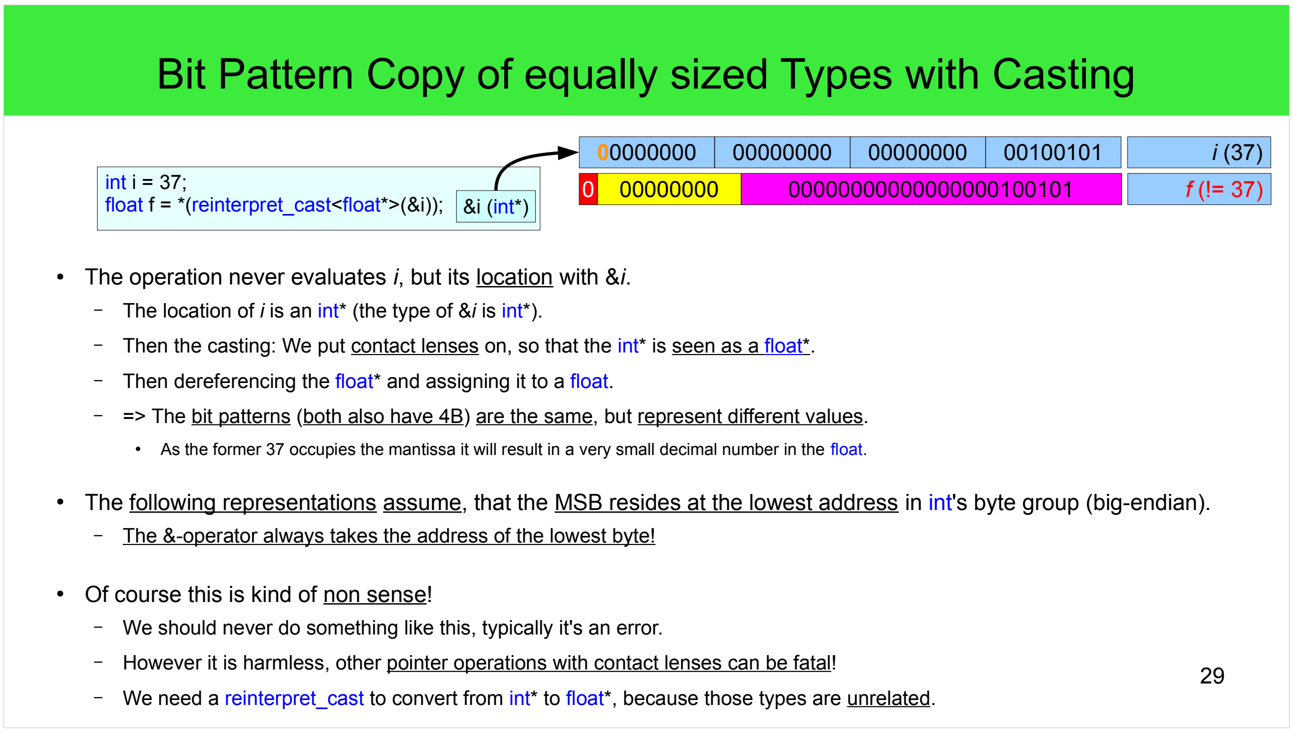
  - In fact <u>there exists no round-to-integer or round-to-floaty function</u> in C/C++!
  - (We can <u>output rounded values</u>.)

```
std::cout<<std::fixed<<std::setprecision(0)<<f<<std::endl;
// >7
```

- (We should not compare <u>calculated</u> floaty type values with the == operator.)
  - Floaty values are just approximate values, we need a kind of "less precision".
  - A comparison can be done with our user defined tolerance (*ACCEPTABLE_ERROR*).

```
float d1 = 9.999999f;
float d2 = 10.f;
bool areEqual = (d1 == d2); // Evaluates to false
const float ACCEPTABLE_ERROR = 0.0001f;
bool areApproxEqual = (std::fabs(d1 - d2) <= ACCEPTABLE_ERROR); // Evaluates to true
```

- There is also an implementation/machine specific constant that denotes the smallest number greater than 1 minus 1. It can be understood as a constant representing the absolute minimal error of floating point operations, it is called epsilon. This constant is present for each floaty type in the special type *std::numeric_limits<double/float>* (in <numeric>), the constant can then be retrieved via *std::numeric_limits<double/float>::epsilon().*
- We didn't talk about infinity on some platforms. We can query the standard library whether the floaty types support infinity: *std::numeric_limits<double/float>::has_infinity*. If it is supported, we can get a special constant representing positive infinity like so: *std::numeric_limits<double/float>::infinity()* and negative infinity (on most platforms) like so: -*std::numeric_limits<double/float>::infinity()*.
- The float-comparison problem is usually not present when comparing constant values, esp. literals.



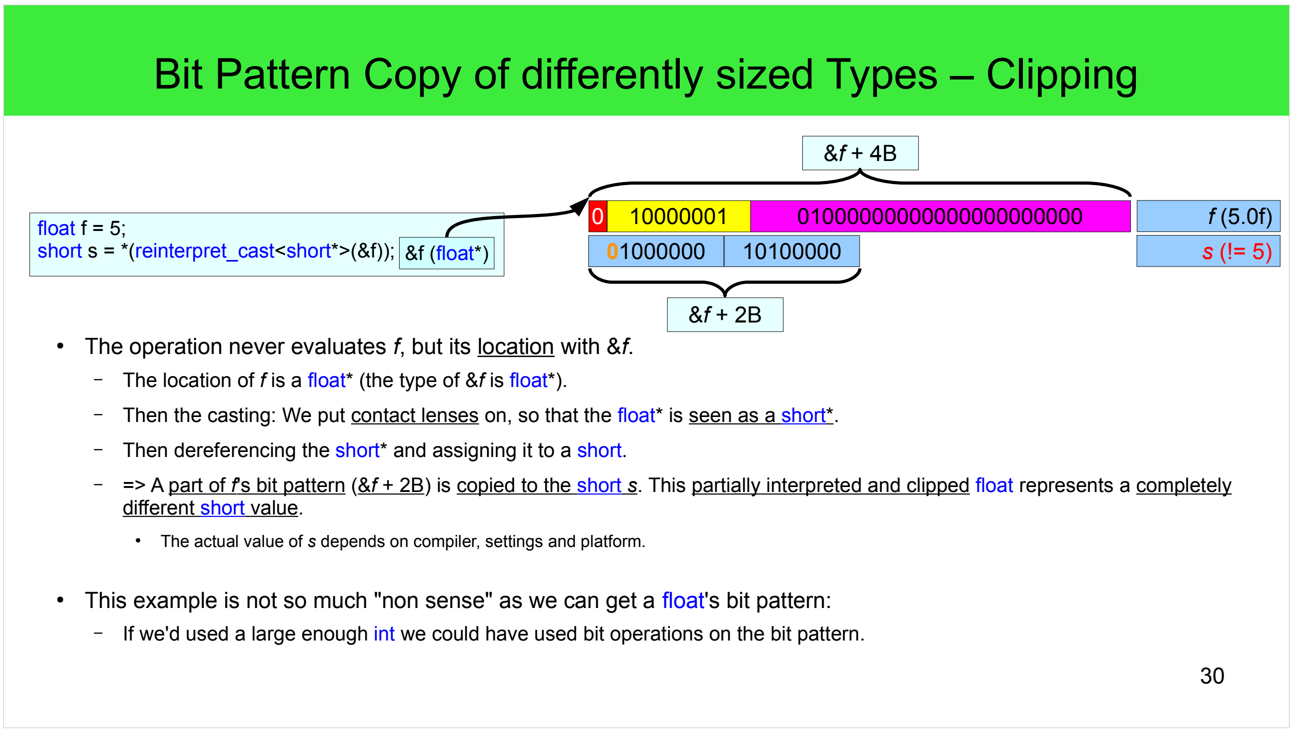
Bit Pattern Copy of equally sized Types with Casting

```
int i = 37;
float f = *(reinterpret_cast<float*>(&i));
```

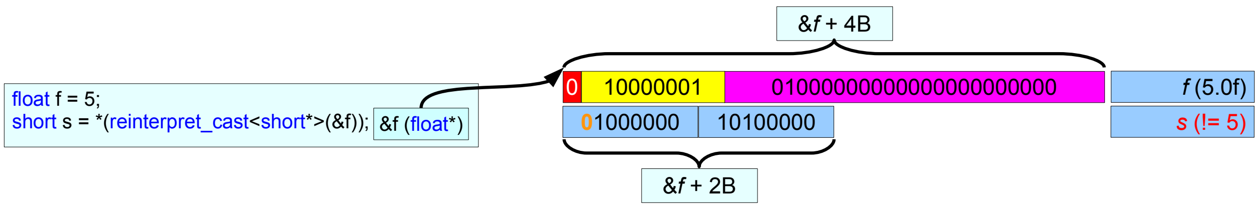
&i (int*)
00000000 | 00000000 | 00000000 | 00100101 | i (37)
0 | 00000000 | 00000000000000000100101 | f (!= 37)

- The operation never evaluates *i*, but its location with &*i*.
  - The location of *i* is an int* (the type of &*i* is int*).
  - Then the casting: We put contact lenses on, so that the int* is seen as a float*.
  - Then dereferencing the float* and assigning it to a float.
  - => The bit patterns (both also have 4B) are the same, but represent different values.
    - As the former 37 occupies the mantissa it will result in a very small decimal number in the float.
- The following representations assume, that the MSB resides at the lowest address in int's byte group (big-endian).
  - The &-operator always takes the address of the lowest byte!
- Of course this is kind of non sense!
  - We should never do something like this, typically it's an error.
  - However it is harmless, other pointer operations with contact lenses can be fatal!
  - We need a reinterpret_cast to convert from int* to float*, because those types are unrelated.

- The casting operation from int* to float* does not move the bits around. All bits remain in their position.
- In this example we could have used the C-notation for casting (convulsive or "wild casting"), in C++ we should use reinterpret_cast<float*>(&*i*). We could not use the static_cast, as it can not convert between unrelated types. The reinterpret_cast is explicitly designed to do a conversion on the bit pattern layer, this makes reinterpret_cast significantly more dangerous than the static_cast! – And therefor the reinterpret_cast is a very ugly syntax!



Bit Pattern Copy of differently sized Types – Clipping



```
float f = 5;
short s = *(reinterpret_cast<short*>(&f));
```

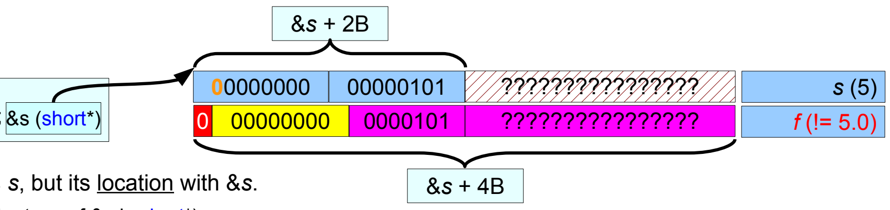
- The operation never evaluates *f*, but its location with &*f*.
  - The location of *f* is a float* (the type of &*f* is float*).
  - Then the casting: We put contact lenses on, so that the float* is seen as a short*.
  - Then dereferencing the short* and assigning it to a short.
  - => A part of *f*'s bit pattern (&*f* + 2B) is copied to the short *s*. This partially interpreted and clipped float represents a completely different short value.
    - The actual value of *s* depends on compiler, settings and platform.
- This example is not so much "non sense" as we can get a float's bit pattern:
  - If we'd used a large enough int we could have used bit operations on the bit pattern.

30

- Notice that bit pattern operations are not allowed on floaty types directly.

Bit Pattern Copy of differently sized Values – Overlapping

```
short s = 5;
float f = *(reinterpret_cast<float*>(&s));
```



- The operation never evaluates *s*, but its location with &*s*.
  - The location of *s* is a short* (the type of &*s* is short*).
  - Then the casting: We put contact lenses on, so that the short* is seen as a float*.
  - Then dereferencing the float* and assigning it to a float.
  - => Not only the bit pattern of *s* is copied into *f*! Also two more bytes right from *s*' bit pattern are copied (&*s* + 4B), these two bytes do not belong to *s*' value!
  - => As we have a completely new bit pattern with partially unknown contents, the value of *f* is also unknown!
    - The actual value of *s* depends on compiler, settings and platform.

Thank you!