Prozedurale Programmierung für Informatiker (PPI)

TUHH
Hamburg University of Technology

Institute for Autonomous Cyber-Physical Systems

Prof. Dr.-Ing. Bernd-Christian Renner
with Dr.-Ing. Peter Oppermann, Wiebke Frenkel, and Johannes Göpfert

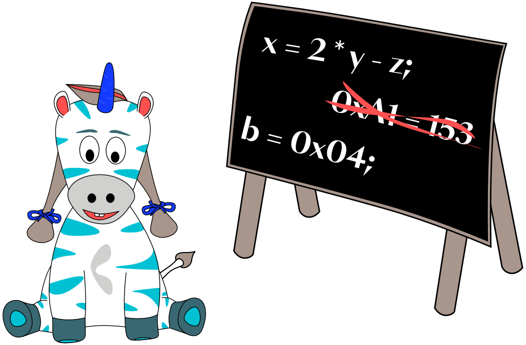
Variables, Numbers and Mathematical Operations

2

Math for Cebras

Learning Goals

- Write programs that deal with numbers
  almost all programs use math in some way
- Make programs flexible
- Interact with a user
- Understand the grammar of C
  and most any procedural language
- Comprehend how computers store numbers
  to avoid errors and write efficient programs



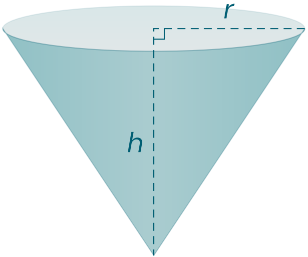
Challenge

Digging a Waterhole

Help Cebra to determine the volume (maximum filling) and the surface area of a waterhole. Derive how many days the waterhole lasts for a given amount of cebras and daily consumption of 4 L.

Write a program that

1. reads the diameter d and height h of a cone plus the number N of cebras
2. calculates the
   - floor space $A = \pi \cdot r^2$
   - volume $V = \frac{1}{3} \cdot \pi \cdot r^2 \cdot h$
   - coverage $C = \frac{V}{N \cdot 4\,\mathrm{L}}$
3. outputs A, V, and C nicely formatted
   - with proper units
   - three decimal digits



Variables, Numbers and Mathematical Operations

2

1

Literals & Variables

Literals

TUHH

Definition: Literal

A number (integer or decimal), a character, or a string in the source code.

Literals

- are part of the program
- cannot be changed during runtime
- have a (data) type
- are converted to binary representation during compilation
  this is extremely important for numbers, because the number in your source code may suffer loss of precision during compilation!

Examples: `-1, 100, 100.0, 1e2, 3.14, "Cebra", "10", 'c', '3'`

Variables

Definition: Variable

A **variable** is a named storage location used to store data in memory.

Each variable has a

- name (the *logical address* of storage location)
- data type
- value (the content of the storage location)
- physical location / address

Properties of variables:

- A program assigns a value to a variable
- Value of a variable
  - can change during program execution
  - can be used by the program

Variable Declaration

- A variable must be *declared* before it is used
  → data type and name
- Variable names
  - consist of letters (`a`–`z`, `A`–`Z`), underscores `_`, and digits `0`–`9`
    - most languages (incl. C) are case-sensitive
    - no German Umlauts or any other characters
    - only the first 31 characters are significant
  - must be unique (in one scope)
  - use camel-casing *or* snake-casing to make variables readable
- Declared variables have *no known value*, they can (read: should always) be *initialized*
  there are exceptions to this … later
- Variables are created (on-the-fly) at runtime

Syntax

```
data_type varName1, varName2, ...;
```

Examples

```c
// declaration only
short a, b;
long height;
int xOffset;    // pick either this
int y_offset;   // *or* this!

// declaration with initialization
long width = 200;
int s = 10, t = 20;
```

Data Types

TUHH

Definition: Data Type

A **data type** determines

- a range of values,
- how to interpret the bits / bytes stored at physical addresses,
- the operations that can be performed on the variables or values.

C knows the data type classes

- boolean (as of C99)
- character
- integer
- floating point
- memory address (pointers)
  we'll have an extra chapter on this later

Boolean Type

Type name	Content	Min. Size	Value Range
`_Bool`	logic value	8 Bit	0 (*false*), 1 (*true*)

The library LIB stdbool.h defines the macros

- `bool` which is an alias of `_Bool`
- `true` with value 1
- `false` with value 0

Remarks

- Formally, `_Bool` belongs to the unsigned integers
- Any value other than 0 and 1 will be converted to 1 (`true`), when treated as a boolean
- The remaining bits of boolean variable are unused (wasted)

Character Type

Type name	*Min. Size*	*Value Range*
`signed char`	8 Bit	-128 … 127
`unsigned char`	8 Bit	0 … 255

Remarks

- Formally, a character type is an integer
- Characters are enclosed in single quotes, e.g., `'A'`, `'!'`, `'x'`
- Non-printable characters have an escape sequence, e.g., `'\n'`, `'\t'`, `'\\'`
- Characters are represented by a number
  values from 0 to 127 are deterministic, remaining values are system-dependent
- The type `char` is implementation-specific
  this only matters, if a `char`-variable is treated as integer

Integer Data Types

Type name	Min. Size	Min. Value Range	
`signed short`	16 Bit	$-2^{15} \dots 2^{15}-1$	(-32 768 … 32 767)
`unsigned short`	16 Bit	$0 \dots 2^{16}-1$	(0 … 65 535)
`signed int`	16 Bit	$-2^{15} \dots 2^{15}-1$	
`unsigned int`	16 Bit	$0 \dots 2^{16}-1$	
`signed long`	32 Bit	$-2^{31} \dots 2^{31}-1$	(-2 147 483 648 … 2 147 483 647)
`unsigned long`	32 Bit	$0 \dots 2^{32}-1$	(0 … 4 294 967 295)
`signed long long`	64 Bit	$-2^{63} \dots 2^{63}-1$	
`unsigned long long`	64 Bit	$0 \dots 2^{64}-1$	

Remarks

- The qualifier `signed` may be omitted
  and it practically always is, if a signed value is required
- The actual size of type `int` is platform-dependent
  the standard only defines a minimum size/range

Floating Point Data Types

TUHH

Type name	*Size*	*Value Range*	*Precision (decimal digits)*
`float`	32 Bit	-3.4e+38 … 3.4e+38	~7
`double`	64 Bit	-1.8e+308 … 1.8e+308	~16

Remarks

- IEC 60559 / IEEE 754 floating-point standard (typically used)
- Symmetric range
- Precision is relative to value (magnitude)
- Special values: **`NaN`** (Not a Number), **`Inf`** (Infinity)
- Can be written as decimal value with an optional exponent (base 10)
  Examples: `25e-1`, `2.5`, `2.5e0`, `0.25e1`
- A type **`long double`** exists, but the standard allows different implementations
  it could even equal **`double`**

First Thoughts about the Waterhole

Reminder

- read d, h (decimals), and N (integer)
- calculate A, V, C (decimals)

waterhole_decl.c

```c
int main(void)
{
  unsigned N;
  double d, h, A, V, C;

  // TODO read from user

  // TODO do the math

  // TODO output

  return 0;
}
```

- ✅ Declare all variables
- ❌ Interact with user to obtain data
- ❌ Solve the problem (math)
- ❌ Output result
- ❌ Write high-quality software

Variables, Numbers and Mathematical Operations

2

2

Expressions

Expression

Definition: Expression

An **expression** is a combination of one or more explicit values, constants, variables, operators, and function results to produce another value.

- If an expression is concluded by a semicolon `;`, it is a statement
- Precondition: Compatible types and operators
  mind the implicit promotion of integers and floats → later
- Expressions are evaluated
  1. based on parentheses-induced order
  2. based on operator precedence
  3. from left-to-right
- Example: `((a * 2.0) + b / 2 - 5) * 1.71`

Assignment Statement

TUHH

Definition: Assignment Statement

An **assignment statement** is used to change the value of a variable.

Syntax

```
varName = expression;
```

- Evaluate `expression` and assign to variable `varName` on left side
  - type of the left side is *not* used for expression evaluation
  - but evaluation result must be compatible
- C converts (*casts*) between data types implicitly and without notice, e.g.,
  - floating point to integer type: truncation of fraction
  - integer to smaller integer: truncation of "higher" bytes
  - signed to unsigned

$\rightarrow$ in general: don't make assumptions or rely on a certain behavior!

Mathematical Operations

Operator	*Operation*	*Operands*	*Example*	*Types*
-	negative sign	variables, literals	`-a`	all
-	subtraction	variables, literals	`a - 7`	all
+	addition	variables, literals	`a + b`	all
*	multiplication	variables, literals	`2.5 * b`	all
/	integer division	variables, literals	`a / b`	all
%	remainder	variables, literals	`a % 2`	integers
++	increment	variables	`a++`	integers
--	decrement	variables	`a--`	integers

- Results of a single expression is that of the larger data type
- There are tons of pitfalls with mathematical expressions
  we talk about that later

Assignments – Short Notations

- Operation on a variable:
  - `i = i + (expr);` → `i += expr;`
  - `i = i - (expr);` → `i -= expr;`
  - `i = i * (expr);` → `i *= expr;`
  - `i = i / (expr);` → `i /= expr;`
- Applicable to all integer and floating-point variables
- Short notation: Expressions are always evaluated first
  parentheses likely required for standard notation
- Example

```
int x = 10;
x *= 2 + 3;  // same as x = x * (2 + 3) = 10 * 5 = 50
```

Progress at the Waterhole

waterhole_no-const.c

```c
int main(void)
{
  unsigned N;
  double d, h, A, V, C;

  // TODO read from user
  // get something for now ...
  d = 3;
  h = 3;
  N = 10;

  A = 3.1415 * d*d / 4;
  V = A * h / 3;
  C = V / (N * 4e-3);

  // TODO output

  return 0;
}
```

- ✅ Declare all variables
- ❌ Interact with user to obtain data
- ✅ Solve the problem (math)
- ❌ Output result
- ❌ Write high-quality software

Variables, Numbers and Mathematical Operations

Constants

2

3

Constants

Definition: Constant

A **constant** in C is either a variable declared as such (i.e., unchangeable) or defined with a preprocessor directive (macro).

Goals:

- Protect values from being changed
- Define values to be used in program in a common, predictable spot
- Prevent *magic numbers*
- Make code more readable and maintainable

Constants in C

- A constant variable
  - is declared with the keyword **`const`** either before or after the type
  - resides in memory (occupies extra space during program execution)
- A constant defined as literal
  - is created with the preprocessor directive **`#define`**
    outside any function and before it is first used
  - does not reside in memory
  - but is inserted (replaced) in all places it occurs at in the code
  - typically used for constants in C
- All constants should be in capital letters
- A plain number in the code is (in many cases) called a *magic number* and should be avoided

Example

```c
const int SCREEN_SIZE = 24;
#define CM_PER_INCH 2.54
...
unsigned screenSize = (unsigned)(CM_PER_INCH * SCREEN_SIZE);  // vs. 2.54 * SCREEN_SIZE (magic number!)
```

A High-Quality Waterhole ... well, you know

TUHH

waterhole_no-io.c

```c
#define PI 3.1415
#define C_PER_DAY 4e-3   // m^3 per day and cebra

int main(void)
{
  unsigned N;
  double d, h, A, V, C;

  // TODO read from user
  // get something for now ...
  d = 3;
  h = 3;
  N = 10;

  A = PI * d*d / 4;
  V = A * h / 3;
  C = V / (N * C_PER_DAY);

  // TODO output

  return 0;
}
```

- ✓ Declare all variables
- ✗ Interact with user to obtain data
- ✓ Solve the problem (math)
- ✗ Output result
- ✓ Write high-quality software

Variables, Numbers and Mathematical Operations

2

4

User Interaction

Creating Output (on the Console)

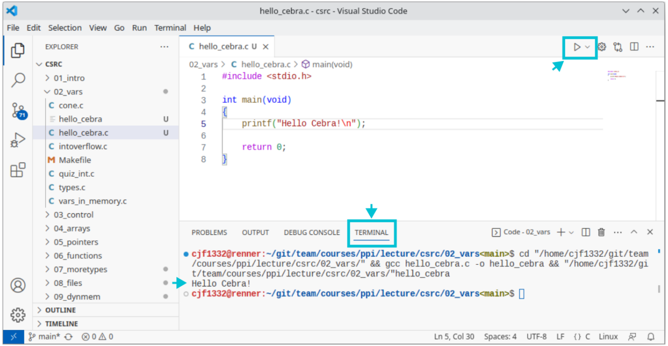
- Writing to the console is achieved with the function `printf`
- It is contained in the library LIB stdio.h, which we must include
  this phrasing is technically not correct, but we leave it at this (for now)
- `printf` is short for "print formatted"

Example

```c
#include <stdio.h>

int main(void)
{
    printf("Hello Cebra!\n")

    return 0;
}
```



(Pretty) Printing to the Console

Syntax

```
printf(format_string, expr1, expr2, ...)
```

- The *`format_string`* is a (string) literal containing text and placeholders
  strings in C are enclosed by double quotes
- A placeholder is actually called *format specifier* and consists of
  1. the letter `%`,
  2. optional formatting fields, and
  3. a data type
- For each placeholder, an expression must be provided
- The order of format specifiers must match the order of expressions in the list
- The type of each format specifier must match the type of the expression

Example:

```
printf("The surface area of the water hole with diameter %d cm is %.3f cm^2\n.", 3, 3.14 * 3 * 3 / 4);
```

Data Type Specifiers and Special Characters (Excerpt)

spec.	*data type*
`d`	signed integer
`u`	unsigned integer
`hd`	short signed integer
`hu`	short unsigned integer
`ld`	long signed integer
`lu`	long unsigned integer
`hx` / `x` / `lx`	hexadecimal representation of an integer
`f` / `lf`	float/double value in fixed-point notation
`e` / `le`	float/double value in exponential form
`c`	character
`s`	string (null-terminated), handled later
`p`	a pointer (a memory address), handled later

escape	*character*
`\n`	newline
`\t`	tab
`\'`	single quotation mark
`\"`	double quotation mark
`\\`	backslash
`%%`	percent

Note: The character `\` is used to *escape* special characters; therefore, it must itself be escaped.

See, e.g., Wikipedia for an exhaustive list of formatting fields.

Reading from the Console

Syntax

```
scanf(format_string, &var1, &var2, ...)
```

- The *`format_string`* is a (string) literal containing placeholders and whitespaces
- A placeholder is called *format specifier* and consists of
  1. the letter `%`,
  2. an optional length (in characters), and
  3. a data type
- All valid characters are converted to the specified type and stored in the variable
- `scanf` stops reading, when the next character is incompatible with stated type
  this may cause infinite loops (see next chapter)!
- The order of the format specifiers must match the order of the variables in the list
- The reason for using the `&` before each variable is discussed later

Example:

```
scanf("%d %f", &age, &height);
```

Getting Done with That Waterhole

waterhole.c

```c
#include <stdio.h>

#define PI 3.1415
#define C_PER_DAY 4e-3   // m^3 per day and cebra

int main(void)
{
  unsigned N;
  double d, h, A, V, C;

  printf("Please enter the waterhole's diameter [m]: ");
  scanf("%lf", &d);
  printf("Please enter the waterhole's depth [m]: ");
  scanf("%lf", &h);
  printf("Please enter the number of cebras: ");
  scanf("%u", &N);

  A = PI * d*d / 4;
  V = A * h / 3;
  C = V / (N * C_PER_DAY);

  printf("\nThe waterhole has a\n"
    "  - floor area of %.3f m^2\n"
    "  - volume of %.3f m^3\n"
    "  - coverage of %.3f d\n\n",
    A, V, C);
  printf("Bye bye!\n\n");

  return 0;
}
```

- Declare all variables
- Interact with user to obtain data
- Solve the problem (math)
- Output result
- Write high-quality software

Variables, Numbers and Mathematical Operations

2

5

A Deeper Look at Numbers

Data Type of Literals

- The data type of a literal only depends on how it is written
  it is **never** affected by context; e.g., assignments
- Without further ado, numeric literals are either of type `int` or `double`
- Integer types other than `int` are defined by a suffix (non-case-sensitive)
  - signed types: `H` (`short`) — `L` (`long`) — `LL` (`long long`)
  - unsigned types: `UH` (`unsigned short`) — `U` (`unsigned int`) — `UL` (`unsigned long`) — `ULL` (`unsigned long long`)
- A numeric literal can be forced to be a floating-point type by adding
  - a fraction (including `.0`)
  - an exponent (including `e0`)
- Floating-point types other than `double` are defined by a trailing (non-case-sensitive)
  - `F` (`float`) — `L` (`long double`)
- Character literals are enclosed in single quotes
- String literals are enclosed in double quotes

Data Type of Expressions

TUHH

- The type of a (binary) expression depends on its operands
- Integer operation, if both operands are integers; otherwise: floating-point
- Operands are promoted to larger (higher rank) type
- Result (of a single operation) is of that larger type
- An overflow occurs, if the result exceeds the range (of the result type)
- Expressions are evaluated one-by-one, independently of other expressions

Expressions and Assignments

- Expressions are evaluated, *before* the (final) result is assigned to a variable
- C converts (*casts*) between data types implicitly and without notice, e.g.,
  - floating point to integer type: truncation of fraction
  - integer to smaller integer: truncation of "higher" bytes
  - signed to unsigned

$\rightarrow$ in general: don't make assumptions or rely on a certain behavior!

A Special Note on Integer Division

TUHH

- An integer division yields an integer result
- A division by zero has undefined behavior
  typically, the program is terminated
- No rounding occurs (fractions are "clipped")
- If rounding is required, it must be done manually

```
int a, b, c;
a = 9;
b = a / 10;        // no rounding, flooring
c = (a + 5) / 10;  // round to nearest (ceiling on tie)
```

Manual rounding with integer arithmetic

to nearest: $(a + d/2)/d \neq a/d + 1/2$

to ceiling: $(a + d - 1)/d \neq a/d + 1 - 1/d$

Pitfalls with Expressions and Assignments

- Problematic assignments (if they remain unnoticed):

```
short x = 7.5;           // loss of precision
unsigned int u = 65536; // will work on some machines only
int i = 2147483648;      // one larger than max int
long l = 2147483648;     // literal is treated as int (success depends on int-size of machine)
float f = 1.234567890;  // loss of precision
```

- Expression is evaluated before assignment

```
double a = 3   / 2;    // integer division, will be 1!
double b = 3   / 2.0;  // float division, will be 1.5!
double c = 3.0 / 2;    // float division, will be 1.5!
```

- Order of execution

```
double a = 6 / 4 / 2;
double b = 6 / 4 / 2.0;
double c = 6.0 / 4 / 2;
```

- Data type ranges

```
short s  = 32767 * 2;        // 32767 == 2^15-1 => works, result?
int   i  = 2147483647 * 2;   // 2147483647 == 2^31-1 => works, result?
long  l  = 2147483647 * 2;   // platform-dependent
long  l2 = 2147483647L * 2;  // platform-dependent
```

Changing the Type of a Variable's Value (Temporarily)

Problem

How to prevent the previous type issue with expressions using only variables?

```
int a, b;
...
double res = a / b;
```

Possible solutions

- ❌ Use final type only (without other reason) → no!
- ❌ Add a bogus operation to expand to appropriate type → never, ever!
- ⊖ Store one operand in the result variable → maybe
- ✅ Change the value of one operand *temporarily* → accepted!
  achieved with explicit type casts, next slide

Explicit Type Casts

Syntax

```
(target_type)(expression)
(target_type)varName
```

Example:

```
int a, b;
double r;
r = (double)a / b;  // cast a's value to double, then apply division
a = (int)(r * b);   // evaluate expression, then cast result to int
```

- The type of an expression (result) can be changed explicitly
- The type of a variable's value can be changed (temporarily)
  i.e., the value stored in a variable is copied and "changed"; the value of the variable stays untouched
- This is called an *explicit type cast*
  - widening: The target type is larger than the original type
    to avoid loss in an expression
  - narrowing: The target type is smaller than the original type
    to fit something into a variable; this may lead to loss
- Has higher precedence than any (binary) operation
  that's what we need the parentheses around the `expression` for

Integer Number Theory

- Humans typically use the decimal system
- Computer memory is organized in blocks of 8 bits
  called byte or octet
- A byte has $2^8 = 256$ states: 0000.0000, 0000.0001, 0000.0010, ..., 1111.1111
- Concise representation via hexadecimal numbers
  digits are 0, 1, ..., 9, A, B, C, D, E, F
- A byte is represented by a 2-digit hexadecimal number: 00, 01, ..., 0F, 10, ..., 1F, 20, ..., FF
  one for higher (left) group of four bits, one for lower (right)
- Prefixes `0x` and `0b` to indicate hexadecimal and binary numbers, respectively

Numbers in different systems

- $1234 = 1024 + 128 + 64 + 16 + 2 = (100.1101.0010)_2 =$ `0b10011010010`
- $1234 = 4 \cdot 256 + 13 \cdot 16 + 2 = (4D2)_{16} =$ `0x4D2`

Integer Representation in Memory

0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

- For an unsigned integer, the value *U* is defined as:

$$U = \sum_{i=0}^{N-1} b_i \cdot 2^i \,, \quad b_i \in \{0, 1\}$$

- For a signed integer, the value *S* is *typically* defined as two's complement:
  C only demands that the range of non-negative values of a signed integer type is a subrange of the corresponding unsigned integer type; the representation of the same value in each type is the same

$$S = -2^{N-1} \cdot b_{N-1} + \sum_{i=0}^{N-2} b_i \cdot 2^i \,, \quad b_i \in \{0, 1\}$$

- In case of the two's complement, inversion is done by flipping all bits and adding 1

Consequences

- The semantic value of a bit pattern in memory depends on its type
  i.e., the same bit pattern may be interpreted differently
- Integers may overflow: e.g., adding 1
  - to the largest unsigned number yields 0
  - adding 1 to the largest signed number produces an undefined result
    a possible outcome is the smallest (most negative) number
- As a result, adding to an integer can make it smaller
- Analogously, multiplications can cause an overflow, too

Advice

- Working with integers can be tricky
- The description of integer arithmetic in the C standard is extensive and beyond the scope of this lecture
- Choose integer variables wisely
  - small enough to save memory
  - large enough to prevent overflow and undefined behavior

Quiz 2.1

Which of the following statements regarding the given bit pattern, representing a signed 8-bit integer using two's complement for negative numbers, is wrong?

1	1	0	0	1	0	0	1
7	6	5	4	3	2	1	0

A The number is odd

B The number is negative

C Multiplication by 2 causes an integer overflow

D Adding 1 doesn't change the number of one-bits

Quiz 2.2

What is the final value of `res`, if an **`unsigned short`** has 2 bytes?

quiz_int.c

```c
#include <stdio.h>

int main(void)
{
  unsigned short res, a;

  a = 0x0080;
  res = a * a;
  res += 2;
  res *= 4;

  printf(”res = %hu\n”, res);
  return 0;
}
```

A 0

B 2

C 8

D $65535 = 2^{16}-1$

ASCII Table

Hex	*…0*	*…1*	*…2*	*…3*	*…4*	*…5*	*…6*	*…7*	*…8*	*…9*	*…A*	*…B*	*…C*	*…D*	*…E*	*…F*
0…	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1…	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2…	SP	!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
3…	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4…	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5…	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6…	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7…	p	q	r	s	t	u	v	w	x	y	z	{	|	}	∼	DEL

Examples

- `'A'` = 0x41 = 65
- `'0'` = 0x30 = 48

Notes

- Characters are numbers
- You can perform calculations; e.g., `'a'`+1 yields `'b'`

Floating-Point Representation

TUHH

$$x = s \cdot m \cdot b^e$$

- Sign s (1 bit)
- Mantissa m (p bits), also called *significand*
  - typically normalized value (one non-zero digit before fraction)
  - smaller values supported by non-normalized mantissa with minimum supported exponent
- Base b, also called *radix*
  $b = 2$ for binary floating point values in IEEE 754
- Exponent e (with size r bits and a bias $B = 2^{r-1} - 1$)

Floating-Point Interpretation (IEEE 754)

S	E	M
1 bit	r bits	p bits

Exponent E	*Mantissa M*	*Interpretation*	*Colloquial*	*Comment*
$E = 0$	$M = 0$	$(-1)^S \times 0$	± 0	Null (denormalized)
$E = 0$	$M > 0$	$(-1)^S \times M/2^p \times 2^{1-B}$	$\pm 0.M \times 2^{1-B}$	denormalized
$0 < E < 2^r - 1$	$M \geq 0$	$(-1)^S \times (1 + M/2^p) \times 2^{E-B}$	$\pm 1.M \times 2^{E-B}$	normalized
$E = 2^r - 1$	$M = 0$	Infinity	$\pm\infty$	Infinity
$E = 2^r - 1$	$M > 0$	Not a Number		Not a Number (NaN)

where $B = 2^{r-1} - 1$ is the exponent bias.

Example of a layout for a 32-bit floating point

1	0	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Issues with Floating-Point Values and Variables

- Do not use floating point variables (or values), if you need integers
- Floating-point operations
  - require a Floating Point Unit (FPU) (as part of the CPU) to be efficient
    this is not the case for many embedded devices
  - are emulated in software, if no FPU exists
    slow and bloats the code produced by the compiler
- There exist different rounding modes
- Adding large and small numbers may yield considerable loss of precision up to: $a + b = a, a \gg b$
- Certain finite decimal numbers are infinite binary numbers; e.g., $\sum_{i=0}^{n-1} 0.1 \neq n \cdot 0.1$

$$0.1 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + \dots$$

Math with Floating-Point Values

Due to the limited precision and construction of floating-point values

- the order of operations in an expression matters
- the type of operation matters
- ...

Mathematical (Associative) Rules are not Guaranteed (from the C standard):

```c
double x, y, z;
/* ... */
x = (x * y) * z;   // not equivalent to x *= y * z;
z = (x - y) + y ;  // not equivalent to z = x;
z = x + x * y;     // not equivalent to z = x * (1.0 + y);
y = x / 5.0;       // not equivalent to y = x * 0.2;
```

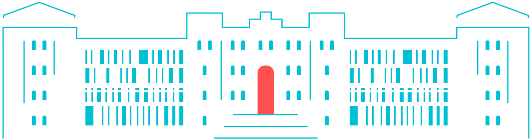
Numbers may not add up "correctly"

```c
double x = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;  // not 0.9!
double y = 2e20 + 1 - 2e20;  // is 0
double z = 2e20 - 2e20 + 1;  // is 1
```

Prozedurale Programmierung für Informatiker (PPI)

TUHH
Hamburg
University of
Technology

Institute for Autonomous Cyber-Physical Systems



Prof. Dr.-Ing. Bernd-Christian Renner
with Dr.-Ing. Peter Oppermann, Wiebke Frenkel, and Johannes Göpfert