8. Floating-point Numbers II

Floating-point Number Systems; IEEE Standard; Limits of Floating-point Arithmetics; Floating-point Guidelines; Harmonic Numbers

Floating-point Number Systems

A Floating-point number system is defined by the four natural numbers:

- $\beta \geq 2$, the base,
- $p \geq 1$, the precision (number of places),
- $e_{\min}$, the smallest possible exponent,
- $e_{\max}$, the largest possible exponent.

Floating-point Number Systems

A Floating-point number system is defined by the four natural numbers:

- $\beta \geq 2$, the base,
- $p \geq 1$, the precision (number of places),
- $e_{\min}$, the smallest possible exponent,
- $e_{\max}$, the largest possible exponent.

Notation:

$$F(\beta, p, e_{\min}, e_{\max})$$

Floating-point number Systems

$F(\beta, p, e_{\min}, e_{\max})$ contains the numbers

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$$

Floating-point number Systems

$F(\beta, p, e_{\min}, e_{\max})$ contains the numbers

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$$

represented in base β:

$$\pm\, d_{0\bullet} d_1 \ldots d_{p-1} \times \beta^e,$$

Floating-point Number Systems

Representations of the decimal number 0.1 (with $\beta = 10$):

$$1.0 \cdot 10^{-1}, \quad 0.1 \cdot 10^{0}, \quad 0.01 \cdot 10^{1}, \quad \ldots$$

Different representations due to choice of exponent

Normalized representation

Normalized number:

$$\pm\, d_0{\bullet}d_1 \ldots d_{p-1} \times \beta^e, \qquad d_0 \neq 0$$

Remark 1

The normalized representation is unique and therefore prefered.

Normalized representation

Normalized number:

$$\pm\, d_0{\bullet}d_1 \dots d_{p-1} \times \beta^e, \qquad d_0 \neq 0$$

Remark 1

The normalized representation is unique and therefore prefered.

Normalized representation

Normalized number:

$$\pm\, d_0 \bullet d_1 \ldots d_{p-1} \times \beta^e, \qquad {\color{red} d_0 \neq 0}$$

Remark 2

The number 0, as well as all numbers smaller than $\beta^{e_{\min}}$, have no normalized representation (we will come back to this later)

Set of Normalized Numbers

$$F^*(\beta, p, e_{\min}, e_{\max})$$

Normalized Representation

Example $F^*(2, 3, -2, 2)$ (only positive numbers)

$d_{0\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalized Representation

Example $F^*(2, 3, -2, 2)$ (only positive numbers)

$d_{0\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalized Representation

Example $F^*(2,3,-2,2)$ (only positive numbers)

$d_{0\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalized Representation

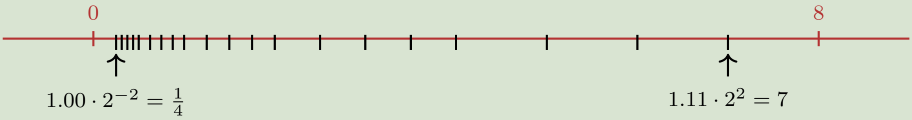
Example $F^*(2, 3, -2, 2)$ (only positive numbers)

$d_{0\bullet}d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Normalized Representation

Example $F^*(2, 3, -2, 2)$ (only positive numbers)

$d_{0\bullet}d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7



Binary and Decimal Systems

- Internally the computer computes with $\beta = 2$ (binary system)

Binary and Decimal Systems

- Internally the computer computes with $\beta = 2$ (binary system)
- Literals and inputs have $\beta = 10$ (decimal system)

Conversion $(0 < x < 2)$

Computation of the *binary representation*:

$$x = \sum_{i=0}^{\infty} b_i 2^{-i}$$

Conversion $(0 < x < 2)$

Computation of the *binary representation*:

$$x = b_{0\bullet}b_1b_2b_3\ldots$$

Conversion $(0 < x < 2)$

Computation of the *binary representation*:

$$\begin{aligned} x &= b_0{\bullet}b_1b_2b_3\ldots \\ &= b_0 + 0{\bullet}b_1b_2b_3\ldots \end{aligned}$$

Conversion $(0 < x < 2)$

Computation of the *binary representation*:

$$
\begin{aligned}
x &= b_{0\bullet}b_1b_2b_3\dots \\
&= b_0 + 0_\bullet b_1b_2b_3\dots \\
&\Longrightarrow
\end{aligned}
$$

Conversion $(0 < x < 2)$

Computation of the *binary representation*:

$$\begin{aligned} x &= b_{0\bullet}b_1b_2b_3\ldots \\ &= b_0 + 0_\bullet b_1b_2b_3\ldots \\ &\Longrightarrow \\ (x - b_0) &= 0_\bullet b_1b_2b_3b_4\ldots \end{aligned}$$

Conversion $(0 < x < 2)$

Computation of the *binary representation*:

$$\begin{aligned} x &= b_0{\bullet}b_1b_2b_3\ldots \\ &= b_0 + 0{\bullet}b_1b_2b_3\ldots \\ &\Longrightarrow \\ 2\cdot(x-b_0) &= b_1{\bullet}b_2b_3b_4\ldots \end{aligned}$$

Conversion $(0 < x < 2)$

Computation of the *binary representation*:

$$
\begin{aligned}
x &= b_{0\bullet}b_1b_2b_3\dots \\
&= b_0 + 0_{\bullet}b_1b_2b_3\dots \\
&\Longrightarrow \\
2\cdot(x-b_0) &= b_{1\bullet}b_2b_3b_4\dots
\end{aligned}
$$

Conversion $(0 < x < 2)$

Computation of the *binary representation*:

$$
\begin{aligned}
x &= b_0{\bullet}b_1 b_2 b_3 \ldots \\
&= b_0 + 0{\bullet}b_1 b_2 b_3 \ldots \\
&\Longrightarrow \\
2 \cdot (x - b_0) &= b_1{\bullet}b_2 b_3 b_4 \ldots
\end{aligned}
$$

```
for (int b_0; x != 0; x = 2 * (x - b_0)) {
  b_0 = (x >= 1);
  std::cout << b_0;
}
```

Example (binary)

$$
\begin{aligned}
x &= {\color{red}1}_\bullet 01011 \\
&= {\color{red}1} + 0_\bullet 01011 \\
&\Longrightarrow \\
2 \cdot (x - {\color{red}1}) &= 0_\bullet 1011
\end{aligned}
$$

Example (binary)

$$x = 1_{\bullet}01011$$
$$= 1 + 0_{\bullet}01011$$
$$\Longrightarrow$$
$$2 \cdot (x - 1) = 0_{\bullet}1011$$

Example (binary)

$$\begin{aligned} x &= 0_{\bullet}1011 \\ &= 0 + 0_{\bullet}1011 \\ &\Longrightarrow \\ 2 \cdot (x - 0) &= 1_{\bullet}011 \end{aligned}$$

Example (binary)

$$
\begin{aligned}
x &= 0_{\bullet}1011 \\
&= 0 + 0_{\bullet}1011 \\
&\Longrightarrow \\
2 \cdot (x - 0) &= 1_{\bullet}011
\end{aligned}
$$

Example (binary)

$$
\begin{aligned}
x &= 1_\bullet 011 \\
&= 1 + 0_\bullet 011 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 0_\bullet 11
\end{aligned}
$$

Example (binary)

$$
\begin{aligned}
x &= 1_\bullet 011 \\
&= 1 + 0_\bullet 011 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 0_\bullet 11
\end{aligned}
$$

Example (binary)

$$
\begin{aligned}
x &= 0_\bullet 11 \\
&= 0 + 0_\bullet 11 \\
&\Longrightarrow \\
2 \cdot (x - 0) &= 1_\bullet 1
\end{aligned}
$$

Example (binary)

$$\begin{aligned} x &= 0_\bullet 11 \\ &= 0 + 0_\bullet 11 \\ &\Longrightarrow \\ 2 \cdot (x - 0) &= 1_\bullet 1 \end{aligned}$$

Example (binary)

$$
\begin{aligned}
x &= 1_{\bullet}1 \\
&= 1 + 0_{\bullet}1 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 1
\end{aligned}
$$

Example (binary)

$$\begin{aligned} x &= 1_\bullet 1 \\ &= 1 + 0_\bullet 1 \\ &\implies \\ 2 \cdot (x - 1) &= 1 \end{aligned}$$

Example (binary)

$$
\begin{aligned}
x &= 1 \\
&= 1 + 0 \\
&\implies \\
2 \cdot (x - 1) &= 0
\end{aligned}
$$

Example (binary)

$$
\begin{aligned}
x &= 1 \\
&= 1 + 0 \\
&\implies \\
2 \cdot (x - 1) &= 0
\end{aligned}
$$

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$		

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$		

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$		

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$		

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$		

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$		

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

$\Rightarrow$ $1.0\overline{0011}$, periodic, *not* finite

Binary Number Representations of 1.1 and 0.1

- are not finite $\Rightarrow$ conversion errors

Binary Number Representations of 1.1 and 0.1

- are not finite $\Rightarrow$ conversion errors
- `1.1f` und `0.1f`: *Approximations* of 1.1 and 0.1

Binary Number Representations of 1.1 and 0.1

- are not finite $\Rightarrow$ conversion errors
- `1.1f` und `0.1f`: *Approximations* of 1.1 and 0.1
- In `diff.cpp`: $1.1 - 1.0 \neq 0.1$

Binary Number Representations of 1.1 and 0.1

on my computer:

$$\begin{aligned} \texttt{1.1} &= \underline{1.10000000000000000}888178\ldots \\ \texttt{1.1f} &= \underline{1.1000000}238418\ldots \end{aligned}$$

Computing with Floating-point Numbers

is nearly as simple as with integers.

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 1.011 \cdot 2^{-1} \end{array}$$

Computing with Floating-point Numbers

Example ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 1.011 \cdot 2^{-1}
\end{array}
$$

1. adjust exponents by denormalizing one number

Computing with Floating-point Numbers

Example ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \checkmark
\end{array}
$$

1. adjust exponents by denormalizing one number

Computing with Floating-point Numbers

Example ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline \end{array}$$

2. binary addition of the significands

Computing with Floating-point Numbers

Example ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 100.101 \cdot 2^{-2} \checkmark \end{array}$$

2. binary addition of the significands

Computing with Floating-point Numbers

Example ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
= & 100.101 \cdot 2^{-2}
\end{array}
$$

3. renormalize

Computing with Floating-point Numbers

Example ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
 & = 1.00101 \cdot 2^{0} \checkmark
\end{array}
$$

3. renormalize

Computing with Floating-point Numbers

Example ($\beta = 2$, $p = 4$):

$$
\begin{array}{rr}
 & 1.111 \cdot 2^{-2} \\
+ & 10.110 \cdot 2^{-2} \\
\hline
 & = 1.00101 \cdot 2^{0}
\end{array}
$$

4. round to p significant places, if necessary

Computing with Floating-point Numbers

Example ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 1.001 \cdot 2^{0} \checkmark \end{array}$$

4. round to p significant places, if necessary

The IEEE Standard 754

defines floating-point number systems and their rounding behavior and is used nearly everywhere

- Single precision (**`float`**) numbers:

  $F^*(2, 24, -126, 127)$ (32 bit)

The IEEE Standard 754

defines floating-point number systems and their rounding behavior and is used nearly everywhere

- Single precision (**`float`**) numbers:

  $F^*(2, 24, -126, 127)$ (32 bit)

- Double precision (**`double`**) numbers:

  $F^*(2, 53, -1022, 1023)$ (64 bit)

The IEEE Standard 754

defines floating-point number systems and their rounding behavior and is used nearly everywhere

- Single precision (**`float`**) numbers:

  $F^*(2, 24, -126, 127)$ (32 bit) plus $0, \infty, \ldots$

- Double precision (**`double`**) numbers:

  $F^*(2, 53, -1022, 1023)$ (64 bit) plus $0, \infty, \ldots$

- All arithmetic operations round the *exact* result to the next representable number

Example: 32-bit Representation of a Floating Point Number

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

$\pm$ Exponent Mantisse

$\pm$

$2^{-126}, \ldots, 2^{127}$

$0, \infty, \ldots$

$1.00000000000000000000000$

$\cdots$

$1.11111111111111111111111$

Rule 1

Do not test rounded floating-point numbers for equality.

Rule 1

Do not test rounded floating-point numbers for equality.

```
for (float i = 0.1; i != 1.0; i += 0.1)
    std::cout << i << "\n";
```

Rule 1

Do not test rounded floating-point numbers for equality.

```
for (float i = 0.1; i != 1.0; i += 0.1)
    std::cout << i << "\n";
```

endless loop because i never becomes exactly 1

Floating-point Rules

Rule 2

Do not add two numbers of very different orders of magnitude!

Rule 2

Do not add two numbers of very different orders of magnitude!

$$\begin{array}{r} 1.000 \cdot 2^5 \\ +1.000 \cdot 2^0 \end{array}$$

Rule 2

Do not add two numbers of very different orders of magnitude!

$$
\begin{aligned}
&1.000 \cdot 2^5 \\
+&1.000 \cdot 2^0 \\
=&\ 1.00001 \cdot 2^5
\end{aligned}
$$

Floating-point Rules

Rule 2

Do not add two numbers of very different orders of magnitude!

$$
\begin{aligned}
&1.000 \cdot 2^5 \\
&+1.000 \cdot 2^0 \\
&= 1.00001 \cdot 2^5 \\
&\text{``}{=}\text{''}\ 1.000 \cdot 2^5 \quad \text{(Rounding on 4 places)}
\end{aligned}
$$

Rule 2

Do not add two numbers of very different orders of magnitude!

$$
\begin{aligned}
& 1.000 \cdot 2^5 \\
& +1.000 \cdot 2^0 \\
& = 1.00001 \cdot 2^5 \\
\text{“}={}\text{”} \; & 1.000 \cdot 2^5 \quad \text{(Rounding on 4 places)}
\end{aligned}
$$

Addition of 1 does not have any effect!

- The n-the harmonic number is

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

- The n-the harmonic number is

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

Harmonic Numbers — Rule 2

- The n-the harmonic number is

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

- This sum can be computed in forward or backward direction, which is mathematically clearly equivalent

Harmonic Numbers

Rule 2

```
std::cout << "Compute H_n for n =? ";
unsigned int n;
std::cin >> n;

float fs = 0;
for (unsigned int i = 1; i <= n; ++i)
    fs += 1.0f / i;
std::cout << "Forward sum = " << fs << "\n";

float bs = 0;
for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;
std::cout << "Backward sum = " << bs << "\n";
```

Harmonic Numbers Rule 2

```
std::cout << "Compute H_n for n =? ";
unsigned int n;
std::cin >> n;

float fs = 0;
for (unsigned int i = 1; i <= n; ++i)
    fs += 1.0f / i;
std::cout << "Forward sum = " << fs << "\n";

float bs = 0;
for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;
std::cout << "Backward sum = " << bs << "\n";
```

Input: **10000000**

forwards: **15.4037**

backwards: **16.686**

Harmonic Numbers

Rule 2

```
std::cout << "Compute H_n for n =? ";
unsigned int n;
std::cin >> n;

float fs = 0;
for (unsigned int i = 1; i <= n; ++i)
    fs += 1.0f / i;
std::cout << "Forward sum = " << fs << "\n";

float bs = 0;
for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;
std::cout << "Backward sum = " << bs << "\n";
```

Input: **100000000**

forwards: **15.4037**

backwards: **18.8079**

Harmonic Numbers

Observation:

- The forward sum stops growing at some point and is “really” wrong.

Harmonic Numbers

Observation:

- The forward sum stops growing at some point and is “really” wrong.
- The backward sum approximates H_n well.

Harmonic Numbers

Observation:

- The forward sum stops growing at some point and is “really” wrong.
- The backward sum approximates H_n well.

Explanation:

Harmonic Numbers

Observation:

- The forward sum stops growing at some point and is “really” wrong.
- The backward sum approximates H_n well.

Explanation:

- For $1 + 1/2 + 1/3 + \cdots$, later terms are too small to actually contribute

Observation:

- The forward sum stops growing at some point and is “really” wrong.
- The backward sum approximates H_n well.

Explanation:

- For $1 + 1/2 + 1/3 + \cdots$, later terms are too small to actually contribute
- Problem similar to $2^5 + 1$ “$=$” 2^5

Rule 4

Do not subtract two numbers with a very similar value.

Cancellation problems, cf. lecture notes.

Literature

David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic (1991)

Randy Glasbergen, 1996

9. Functions I

Defining and Calling Functions, Evaluation of Function Calls, the Type **`void`**

Computing Powers

```
double a;
int n;
std::cin >> a; // Eingabe a
std::cin >> n; // Eingabe n

double result = 1.0;
if (n < 0) { // a^n = (1/a)^(-n)
  a = 1.0/a;
  n = -n;
}
for (int i = 0; i < n; ++i)
  result *= a;

std::cout << a << "^" << n << " = " << result << ".\n";
```

Computing Powers

```
double a;
int n;
std::cin >> a; // Eingabe a
std::cin >> n; // Eingabe n

double result = 1.0;
if (n < 0) { // a^n = (1/a)^(-n)
  a = 1.0/a;
  n = -n;
}
for (int i = 0; i < n; ++i)
  result *= a;

std::cout << a << "^" << n << " = " << result << ".\n";
```

Computing Powers

```
double a;
int n;
std::cin >> a; // Eingabe a
std::cin >> n; // Eingabe n

double result = 1.0;
if (n < 0) { // a^n = (1/a)^(-n)
  a = 1.0/a;
  n = -n;
}
for (int i = 0; i < n; ++i)
  result *= a;

std::cout << a << "^" << n << " = " << pow(a,n) << ".\n";
```

"Funktion pow"

Function to Compute Powers

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```

Function to Compute Powers

```
double pow(double b, int e){...}
```

Function to Compute Powers

```
// Prog: callpow.cpp
// Define and call a function for computing powers.
#include <iostream>

double pow(double b, int e){...}

int main()
{
  std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
  std::cout << pow( 1.5, 2) << "\n"; // outputs 2.25
  std::cout << pow(-2.0, 9) << "\n"; // outputs -512

  return 0;
}
```

Function Definitions

T fname (T_1 pname$_1$, T_2 pname$_2$, ..., T_N pname$_N$)
    block

function name

Function Definitions

return type

T fname (T_1 pname$_1$, T_2 pname$_2$, $\ldots$, T_N pname$_N$)
    block

function name

Function Definitions

return type

T fname (T_1 pname$_1$, T_2 pname$_2$, . . . ,T_N pname$_N$)
block

function name

formal arguments

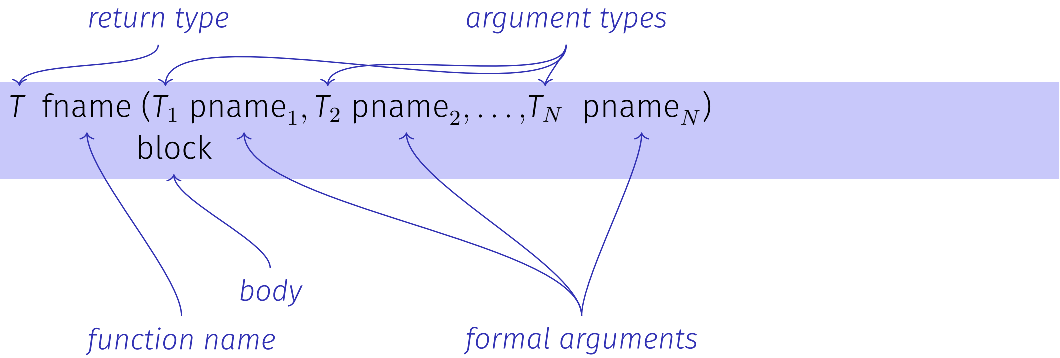
Function Definitions

return type *argument types*

T fname (T_1 pname$_1$, T_2 pname$_2$, ..., T_N pname$_N$)
block

function name *formal arguments*

Function Definitions



Xor

```
// post: returns l XOR r
bool Xor(bool l, bool r)
{
    return l && !r || !l && r;
}
```

Harmonic

```
// PRE: n >= 0
// POST: returns nth harmonic number
//       computed with backward sum
float Harmonic(int n)
{
    float res = 0;
    for (unsigned int i = n; i >= 1; --i)
        res += 1.0f / i;
    return res;
}
```

min

```
// POST: returns the minimum of a and b
int min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
}
```

Function Calls

fname (*expression*$_1$, *expression*$_2$, …, *expression*$_N$)

- All call arguments must be convertible to the respective formal argument types.

Function Calls

fname (*expression*$_1$, *expression*$_2$, $\ldots$, *expression*$_N$)

- All call arguments must be convertible to the respective formal argument types.
- The function call is an expression of the return type of the function.

Function Calls

fname (*expression*$_1$, *expression*$_2$, …, *expression*$_N$)

- All call arguments must be convertible to the respective formal argument types.
- The function call is an expression of the return type of the function.

Example: **`pow(a,n)`**: Expression of type **`double`**

Function Calls

For the types we know up to this point it holds that:

- Call arguments are R-values
  $\hookrightarrow$ *call-by-value* (also *pass-by-value*), more on this soon
- The function call is an R-value.

Function Calls

For the types we know up to this point it holds that:

- Call arguments are R-values
  $\hookrightarrow$ *call-by-value* (also *pass-by-value*), more on this soon
- The function call is an R-value.

fname: R-value $\times$ R-value $\times \cdots \times$ R-value $\longrightarrow$ R-value

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Call of pow

Evaluation Function Call

```
double pow(double b, int e){          b=2.0,e=-2
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){          b=2.0,e=-2
    assert (e >= 0 || b != 0);        // ok
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

result=1.0

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

e == -2

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;        → b=0.5
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

e=2

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)        → i=0
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)      → i=0
        result * = b;                 → result=0.5
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)  → i=1
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)     → i=1
        result * = b;                → result=0.25
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)          → i=2
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;        → result=0.25
}

...
pow (2.0, -2)
```

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

result=0.25

Return

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Return

value: 0.25

Evaluation Function Call

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

→ value: 0.25

Scope of Formal Arguments

```
int main(){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // -2
    return 0;
}
```

Scope of Formal Arguments

```
double pow(double b, int e){
    double r = 1.0;
    if (e<0) {
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        r * = b;
    return r;
}
```

```
int main(){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // -2
    return 0;
}
```

Scope of Formal Arguments

```
double pow(double b, int e){
    double r = 1.0;
    if (e<0) {
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        r * = b;
    return r;
}
```

```
int main(){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // -2
    return 0;
}
```

Not the formal arguments `b` and `e` of pow but the variables defined here locally in the body of `main`

The type `void`

```
// POST: "(i, j)" has been written to standard output
???? print_pair(int i, int j) {
    std::cout << "(" << i << ", " << j << ")\n";
}

int main() {
    print_pair(3,4); // outputs (3, 4)
    return 0;
}
```

The type `void`

```
// POST: "(i, j)" has been written to standard output
void print_pair(int i, int j) {
    std::cout << "(" << i << ", " << j << ")\n";
}

int main() {
    print_pair(3,4); // outputs (3, 4)
    return 0;
}
```

The type `void`

- Fundamental type with empty value range

The type `void`

- Fundamental type with empty value range
- Usage as a return type for functions that do *only* provide an effect

`void`-Functions

- do not require **`return`**.
- execution ends when the end of the function body is reached or if
- **`return;`** is reached

Functions and `return`

Wrong:

```
bool compare(float x, float y) {
  float delta = x - y;
  if (delta*delta < 0.001f) return true;
}
```

Functions and `return`

The behavior of a function with non-`void` return type is **undefined** if the end of the function body is reached without a `return` statement.

Wrong:

```
bool compare(float x, float y) {
  float delta = x - y;
  if (delta*delta < 0.001f) return true;
}
```

Here the value of `compare(10,20)` is undefined.

Functions and `return`

The behavior of a function with non-`void` return type is **undefined** if the end of the function body is reached without a `return` statement.

Better:

```
bool compare(float x, float y) {
   float delta = x - y;
   if (delta*delta < 0.001f)
    return true;
   else
    return false;
}
```

All execution paths reach a `return`

Functions and `return`

The behavior of a function with non-`void` return type is **undefined** if the end of the function body is reached without a `return` statement.

Even better and simpler

```
bool compare(float x, float y) {
   float delta = x - y;
   return delta*delta < 0.001f;
}
```