

## 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.

Notation:

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

# Floating-point number Systems

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

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

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

represented in base  $\beta$ :

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

# Floating-point Number Systems

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

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

Different representations due to choice of exponent

# Normalized representation

Normalized number:

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

## Remark 1

The normalized representation is unique and therefore preferred.

## 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

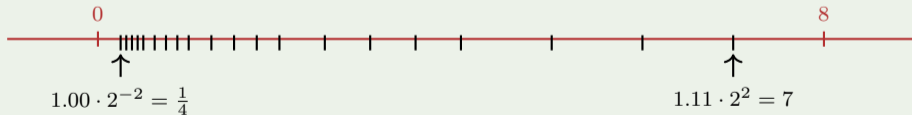
$$F^*(\beta, p, e_{\min}, e_{\max})$$

# Normalized Representation

Example  $F^*(2, 3, -2, 2)$

(only positive numbers)

$d_0.d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
$1.00_2$	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)
- Literals and inputs have  $\beta = 10$   
(decimal system)
- Inputs have to be converted!



# Conversion Decimal $\rightarrow$ Binary

Assume,  $0 < x < 2$ .

Binary representation:

$$\begin{aligned}x &= \sum_{i=-\infty}^0 b_i 2^i = b_0 \bullet b_{-1} b_{-2} b_{-3} \dots \\&= b_0 + \sum_{i=-\infty}^{-1} b_i 2^i = b_0 + \sum_{i=-\infty}^0 b_{i-1} 2^{i-1} \\&= b_0 + \underbrace{\left( \sum_{i=-\infty}^0 b_{i-1} 2^i \right)}_{x' = b_{-1} \bullet b_{-2} b_{-3} b_{-4}} / 2\end{aligned}$$

# Conversion Decimal $\rightarrow$ Binary

Assume  $0 < x < 2$ .

■ Hence:  $x' = b_{-1}b_{-2}b_{-3}b_{-4}\dots = 2 \cdot (x - b_0)$

■ Step 1 (for  $x$ ): Compute  $b_0$ :

$$b_0 = \begin{cases} 1, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

■ Step 2 (for  $x$ ): Compute  $b_{-1}, b_{-2}, \dots$ :

Go to step 1 (for  $x' = 2 \cdot (x - b_0)$ )

# 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
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

$\Rightarrow 1.000\overline{11}$ , periodic, *not* finite

# Binary Number Representations of 1.1 and 0.1

- are not finite, hence there are errors when converting into a (finite) binary floating-point system.
- `1.1f` and `0.1f` do not equal 1.1 and 0.1, but are slightly inaccurate approximation of these numbers.
- In `diff.cpp`:  $1.1 - 1.0 \neq 0.1$

# Binary Number Representations of 1.1 and 0.1

on my computer:

$$\begin{aligned} \mathbf{1.1} &= \underline{1.1000000000000000}888178\dots \\ \mathbf{1.1f} &= \underline{1.1000000}238418\dots \end{aligned}$$

# Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 1.011 \cdot 2^{-1} \\ \hline = 1.001 \cdot 2^0 \end{array}$$

1. adjust exponents by denormalizing one number
2. binary addition of the significands
3. renormalize
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:

$$F^*(2, 24, -126, 127) \text{ (32 bit)} \quad \text{plus } 0, \infty, \dots$$

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

$$F^*(2, 53, -1022, 1023) \text{ (64 bit)} \quad \text{plus } 0, \infty, \dots$$

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

# The IEEE Standard 754

Why

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

- 1 sign bit
- 23 bit for the significand (leading bit is 1 and is not stored)
- 8 bit for the exponent (256 possible values)(254 possible exponents, 2 special values: 0,  $\infty$ ,...)

⇒ 32 bit in total.



# The IEEE Standard 754

Why

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

- 1 sign bit
- 52 bit for the significand (leading bit is 1 and is not stored)
- 11 bit for the exponent (2046 possible exponents, 2 special values: 0,  $\infty, \dots$ )

$\Rightarrow$  64 bit in total.

# Example: 32-bit Representation of a Floating Point Number



± Exponent

Mantisse

±  $2^{-126}, \dots, 2^{127}$   
 $0, \infty, \dots$

1.000000000000000000000000  
...  
1.111111111111111111111111

### 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

### Rule 2

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

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

Addition of 1 does not have any effect!

- The  $n$ -th 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

---

```
// Program: harmonic.cpp
// Compute the n-th harmonic number in two ways.

#include <iostream>

int main()
{
    // Input
    std::cout << "Compute H_n for n =? ";
    unsigned int n;
    std::cin >> n;

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

    // Backward sum
    float bs = 0;
    for (unsigned int i = n; i >= 1; --i)
        bs += 1.0f / i;

    // Output
    std::cout << "Forward sum = " << fs << "\n"
              << "Backward sum = " << bs << "\n";

    return 0;
}
```

---

# Harmonic Numbers

## Rule 2

Results:



```
Compute H_n for n =? 10000000  
Forward sum = 15.4037  
Backward sum = 16.686
```



```
Compute H_n for n =? 100000000  
Forward sum = 15.4037  
Backward sum = 18.8079
```

# Harmonic Numbers

## Rule 2

Observation:

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

Explanation:

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



### 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**

# Functions

- encapsulate functionality that is frequently used (e.g. computing powers) and make it easily accessible
- structure a program: partitioning into small sub-tasks, each of which is implemented as a function

⇒ Procedural programming; procedure: a different word for function.

# Example: 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;
```



"Funktion pow"

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

# 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

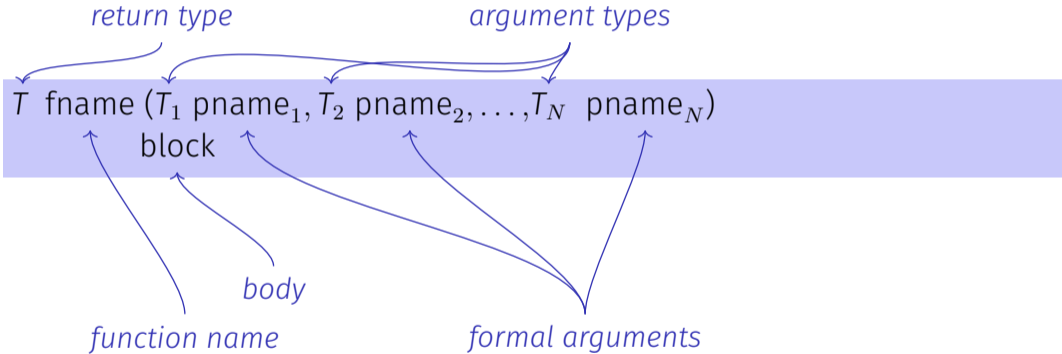
```
// Prog: callpow.cpp  
// Define and call a function for computing powers.
```

```
#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





# Defining Functions

- may not occur *locally*, i.e. not in blocks, not in other functions and not within control statements
- can be written consecutively without separator in a program

```
double pow (double b, int e)
```

```
{
```

```
    ...
```

```
}
```

```
int main ()
```

```
{
```

```
    ...
```

```
}
```

## Example: Xor

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

## Example: 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;
}
```

# Example: 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

$fname ( expression_1, expression_2, \dots, expression_N )$

- 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. Value and effect as given in the postcondition of the function *fname*.

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.

$fname: \text{R-value} \times \text{R-value} \times \dots \times \text{R-value} \longrightarrow \text{R-value}$

# Evaluation of a Function Call

- Evaluation of the call arguments
- Initialization of the formal arguments with the resulting values
- Execution of the function body: formal arguments behave like local variables
- Execution ends with **return** *expression*;

Return value yields the value of the function call.

# Example: Evaluation Function Call

The diagram illustrates the evaluation of a function call. A red arrow labeled "Call of pow" points from the function call `pow(2.0, -2)` at the bottom to the function definition above. Another red arrow labeled "Return" points from the `return result;` line back to the 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)
```



# sometimes em formal arguments

- Declarative region: function definition
- are *invisible* outside the function definition
- are allocated for each call of the function (automatic storage duration)
- modifications of their value do not have an effect to the values of the call arguments (call arguments are R-values)

## 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
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
- 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  
or
- **return** *expression*; is reached.

Expression with type **void** (e.g. a call of a function with return type **void**)

# 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;  
}
```