6. 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),
- $e_{\text{min}}$, the smallest possible exponent,
- $e_{\text{max}}$, the largest possible exponent.

Notation:

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

Floating-point number Systems

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

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

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

Represented in base $\beta$:

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

Example

- $\beta = 10$

Representations of the decimal number 0.1

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

Normalized number:

\[ \pm d_0 \cdot d_1 \ldots d_{p-1} \times \beta^e, \quad d_0 \neq 0 \]

**Remark 1**
The normalized representation is unique and therefore preferred.

**Remark 2**
The number 0 (and all numbers smaller than \( \beta^{e_{\text{min}}} \)) have no normalized representation (we will deal with this later)!

Set of Normalized Numbers

\[ F^*(\beta, p, e_{\text{min}}, e_{\text{max}}) \]

Normalized Representation

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

<table>
<thead>
<tr>
<th>( d_0 \cdot d_1 d_2 )</th>
<th>( e = -2 )</th>
<th>( e = -1 )</th>
<th>( e = 0 )</th>
<th>( e = 1 )</th>
<th>( e = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00_2</td>
<td>0.25</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1.01_2</td>
<td>0.3125</td>
<td>0.625</td>
<td>1.25</td>
<td>2.5</td>
<td>5</td>
</tr>
<tr>
<td>1.10_2</td>
<td>0.375</td>
<td>0.75</td>
<td>1.5</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1.11_2</td>
<td>0.4375</td>
<td>0.875</td>
<td>1.75</td>
<td>3.5</td>
<td>7</td>
</tr>
</tbody>
</table>

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 → Binary

Assume $0 < x < 2$.

- Hence: $x' = b_{-1} b_{-2} b_{-3} b_{-4} \ldots = 2 \cdot (x - b_0)$
- 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 $b_{-1}, b_{-2}, \ldots$
  Go to step 1 (for $x' = 2 \cdot (x - b_0)$)

Binary representation of 1.1

<table>
<thead>
<tr>
<th>$x$</th>
<th>$b_i$</th>
<th>$x - b_i$</th>
<th>$2(x - b_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>$b_0 = 1$</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>0.2</td>
<td>$b_{-1} = 0$</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>0.4</td>
<td>$b_{-2} = 0$</td>
<td>0.4</td>
<td>0.8</td>
</tr>
<tr>
<td>0.8</td>
<td>$b_{-3} = 0$</td>
<td>0.8</td>
<td>1.6</td>
</tr>
<tr>
<td>1.6</td>
<td>$b_{-4} = 1$</td>
<td>0.6</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2</td>
<td>$b_{-5} = 1$</td>
<td>0.2</td>
<td>0.4</td>
</tr>
</tbody>
</table>

⇒ $1.00011$, 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:

1.1 = 1.1000000000000000888178…
1.1f = 1.1000000238418…
The Excel-2007-Bug

std::cout << 850 * 77.1; // 65535

- 77.1 does not have a finite binary representation, we obtain 65534.9999999999927...
- For this and exactly 11 other “rare” numbers the output (and only the output) was wrong.

Computing with Floating-point Numbers

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

\[
\begin{align*}
1.111 \cdot 2^{-2} &+ 1.011 \cdot 2^{-1} \\
&= 1.001 \cdot 2^0
\end{align*}
\]

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
- is used nearly everywhere
- Single precision (float) numbers:
  \[ F^*(2, 24, -126, 127) \] plus 0, $\infty$, ...
- Double precision (double) numbers:
  \[ F^*(2, 53, -1022, 1023) \] plus 0, $\infty$, ...
- All arithmetic operations round the exact result to the next representable number

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, $\infty$, ...)

$\Rightarrow$ 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, \( \infty \), . . .)

⇒ 64 bit in total.

Floating-point Rules

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!

```
1.000 \cdot 2^5
+1.000 \cdot 2^0
= 1.00001 \cdot 2^5
```

“≈” 1.000 \cdot 2^5 (Rounding on 4 places)

Addition of 1 does not have any effect!

Harmonic Numbers

Rule 2

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.
Harmonic Numbers

Rule 2

// 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 << 

    << "Backward sum = " << bs << "\n";
    return 0;
}

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

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 $2^5 + 1 "=" 2^5$

Floating-point Guidelines

Rule 3

Rule 4

Do not subtract two numbers with a very similar value.

Cancellation problems, cf. lecture notes.
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;

std::cout << a << "^" << n << " = " << resultpow(a,n) << ".n;"

7. Functions I

Defining and Calling Functions, Evaluation of Function Calls, the Type void, Pre- and Post-Conditions
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 Definitions

T fname (T1 pname1, T2 pname2, ..., TN pnameN)
block
function name
return type
body
formal arguments
argument types
return type

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

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

Example: Harmonic

```c
// 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

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

Function Calls

```c
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. 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
- The function call is an R-value.

**fname:** R-value × R-value × ... × R-value → 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

```c
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;
}
```

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**

```c
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;
}
```

Not the formal arguments \( b \) and \( e \) of \( \text{pow} \) but the variables defined here locally in the body of \( \text{main} \)

---

**The type void**

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

```c
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;
}
```

---

**void-Functions**

- do not require \texttt{return}.
- execution ends when the end of the function body is reached or if \texttt{return} is reached
- \texttt{return expression} is reached.

Expression with type \texttt{void} (e.g. a call of a function with return type \texttt{void})

---

**Pre- and Postconditions**

- characterize (as complete as possible) what a function does
- document the function for users and programmers (we or other people)
- make programs more readable: we do not have to understand \textit{how} the function works
- are ignored by the compiler
- Pre and postconditions render statements about the correctness of a program possible – provided they are correct.
Preconditions

precondition:
- what is required to hold when the function is called?
- defines the domain of the function

$0^e$ is undefined for $e < 0$

// PRE: $e \geq 0 \, || \, b \neq 0.0$

Postconditions

postcondition:
- What is guaranteed to hold after the function call?
- Specifies value and effect of the function call.

Here only value, no effect.

// POST: return value is $b^e$

Pre- and Postconditions

- should be correct:
  - if the precondition holds when the function is called then also the postcondition holds after the call.

Funktion \texttt{pow}: works for all numbers $b \neq 0$

Pre- and Postconditions

- We do not make a statement about what happens if the precondition does not hold.
  - C++-standard-slang: "Undefined behavior".

Function \texttt{pow}: division by 0
Pre- and Postconditions

- pre-condition should be as *weak* as possible (largest possible domain)
- post-condition should be as *strong* as possible (most detailed information)

White Lies...

```cpp
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

is formally incorrect:

- Overflow if e or b are too large
- \( b^e \) potentially not representable as a double (holes in the value range!)

White Lies are Allowed

```cpp
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

The exact pre- and postconditions are platform-dependent and often complicated. We abstract away and provide the mathematical conditions. ⇒ compromise between formal correctness and lax practice.

Checking Preconditions...

- Preconditions are only comments.
- How can we ensure that they hold when the function is called?
# Postconditions with Asserts

- The result of “complex” computations is often easy to check.
- Then the use of asserts for the postcondition is worthwhile.

```cpp
double pow(double b, int e) {
    assert (e >= 0 || b != 0);
    double result = 1.0;
    ...
}
```

### Exceptions

- Assertions are a rough tool; if an assertions fails, the program is halted in an unrecoverable way.
- C++ provides more elegant means (exceptions) in order to deal with such failures depending on the situation and potentially without halting the program.
- Failsafe programs should only halt in emergency situations and therefore should work with exceptions. For this course, however, this goes too far.