ZÁPADOČESKÁ
UNIVERZITA
V PLZNI

Documentation

Multiple-Precision Integer Calculator

Gaétan Thomassin
KIV/PC Semestral Work

December 24, 2025

Contents

1 Assignment

The objective of this project is to implement a console-based calculator capable of operating on integers of arbitrary precision. The required functionality includes:

- An internal multiple-precision signed integer type with dynamic memory management.
- Parsing of integer literals in decimal, binary (optionally with `0b` prefix), and hexadecimal (`0x` prefix), including two's-complement interpretation for binary/hex inputs.
- Arithmetic operators: addition, subtraction, multiplication, integer division, remainder, exponentiation, and factorial.
- Correct printing of results in decimal, binary, and hexadecimal formats; binary/hex printing must use the minimal two's-complement width consistent with the value's sign.
- A REPL (Read–Eval–Print Loop) supporting commands to switch output formats ("bin", "hex", "dec", "out") and to process expressions from files.
- ANSI C90 compatibility.

2 Analysis of the problem

This section outlines the problem space, enumerates candidate approaches, and justifies the design decisions chosen for the final implementation. The goal is to make the reasoning in approaching the problem explicit, list possible implementation choices and explain why particular algorithms and data structures were selected over others.

2.1 Objectives and constraints

The primary goals are:

- **Correctness:** Arithmetic behavior (signed addition/subtraction, multiplication, integer division, remainder, exponentiation, and factorial) must be mathematically correct and consistent with documented semantics (e.g., remainder sign rules, two's-complement input interpretation).
- **Portability:** The code must compile under ANSI C90 on typical 32-bit and 64-bit platforms.
- **Performance:** Algorithms should perform efficiently for the range of inputs expected in the assignment (e.g. around five operands in the range of 10^{40} based on the examples provided in the assignment, or smaller operands with cumulated factorial/exponentiation operations). While extreme asymptotic performance for very large operands is desirable, it is secondary to correctness and clarity.
- **Simplicity and maintainability:** The implementation should be understandable and extensible.

Important constraints:

- The code must not assume a specific machine endianness for correctness, though it may rely on word-size properties for performance optimizations.
- No external big-integer libraries are permitted; the solution must be implemented entirely in ANSI C.
- Memory safety and clear ownership semantics are mandatory (explicit initialization/freeing).

2.2 Representation choices

Multiple-precision integers can be represented in several ways. The two principal categories are:

Base-256 (byte array) representation: This represents the integer as an array of bytes where each element stores a value between 0 and 255, and carries are propagated base-256. This is simple and compact but inefficient regarding native machine word multipliers, as each arithmetic operation requires more limbs and carry operations per calculation.

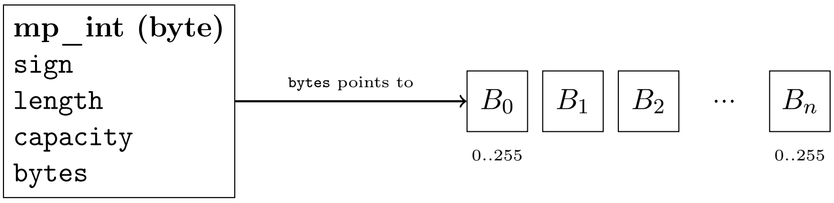

Figure 1: Base-256 Representation (Byte Array)

Limb-based representation (machine-word): This uses an array of "limbs" where each limb is a word of system-dependent size (e.g., 32-bit) or a chosen half-size (e.g., 16-bit). Arithmetic is performed using a wider accumulator (double-width) to hold intermediate products.

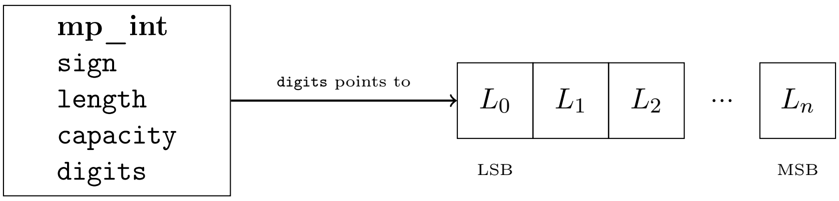

Figure 2: Limb-based representation (Little-Endian storage)

Chosen approach: A limb-based representation with compile-time selection of limb width. Depending on the system's integer size, the code selects either 32-bit limbs (if `unsigned long` is 64-bit) or 16-bit limbs (if `unsigned long` is 32-bit). This ensures portability across 32-bit and 64-bit platforms while guaranteeing that the product of two limbs fits within the accumulator type.

2.3 Arithmetic algorithm choices

2.3.1 Addition and subtraction

Implementing addition and subtraction for arbitrary-precision integers typically follows one of two approaches regarding sign handling:

- **Two's Complement:** The entire number is stored in two's complement. This simplifies the addition logic (no separate subtraction path), but complicates resizing operations and detection of overflow for arbitrary bit-widths.
- **Sign-Magnitude:** The sign is stored separately from the absolute value (magnitude). This requires separate logic for adding magnitudes (when signs match) and subtracting magnitudes (when signs differ), but simplifies the underlying multi-word algorithms which can treat all limbs as unsigned.

Chosen approach: The implementation uses the **Sign-Magnitude** approach. Addition and subtraction on limb arrays are implemented via standard carry/borrow propagation on absolute values, combined with sign logic:

- If signs are equal: Add magnitudes and preserve the sign.
- If signs differ: Subtract the smaller magnitude from the larger and apply the sign of the larger magnitude.

This design is robust, easy to test, and avoids the complexity of in-place sign-aware mutations.

2.3.2 Multiplication

Possible algorithms include:

- **Naive (schoolbook):** $O(N \times M)$ limb multiplications. Simple with low constant overhead; ideal for small operand sizes.
- **Karatsuba:** Asymptotically $O(N^{\log_2 3}) \approx O(N^{1.58})$. It improves performance for moderately large operands but introduces overhead (splitting, recursion, temporary allocations).
- **Toom–Cook / FFT-based:** These offer further asymptotic improvements but come with substantial implementation complexity and memory overhead, making them suitable only for extremely large operands.

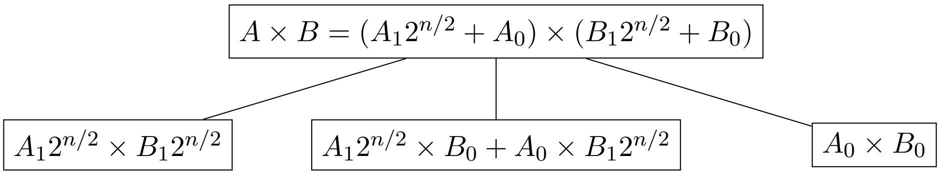
$A \times B = (A_1 2^{n/2} + A_0) \times (B_1 2^{n/2} + B_0)$
$A_1 2^{n/2} \times B_1 2^{n/2}$
$A_1 2^{n/2} \times B_0 + A_0 \times B_1 2^{n/2}$
$A_0 \times B_0$

Figure 3: Concept of Karatsuba Multiplication (Split & Recurse) for Binary Numbers

Chosen approach: A hybrid implementation. The system uses naive multiplication for small numbers (below a defined threshold) and switches to Karatsuba multiplication for larger operands to balance low overhead with asymptotic speedup.

2.3.3 Division

Candidate designs:

- **Knuth's Algorithm D (normalized long division):** Numerically robust and efficient for multi-word division. It works by normalizing the divisor (shifting it so the most significant bit is set) to allow precise estimation of the next quotient digit using only the top few limbs. A correction step adjusts the estimate if necessary.
- **Shift and Subtract (Restoring Division):** Conceptually simpler—the divisor is aligned via shifts and then repeatedly doubled and subtracted to reconstruct the quotient.

Trade-offs and choice: While Algorithm D provides better theoretical performance per digit, it is complex to implement correctly (requiring normalization and correction loops). The Shift and Subtract method was chosen for its implementation simplicity and robustness, which is sufficient for the project's performance requirements.

2.4 Exponentiation and factorial

Exponentiation Calculating a^b can be approached in several ways:

- **Naive Iteration:** Multiplying the base a by itself $b-1$ times. This is $O(b)$ multiplications, which is prohibitively slow for large exponents.

- **Binary Exponentiation (Square-and-Multiply):** This method scans the bits of the exponent. The base is squared at each step, and multiplied into the result only when the current exponent bit is 1. This reduces the number of multiplications to $O(\log b)$.

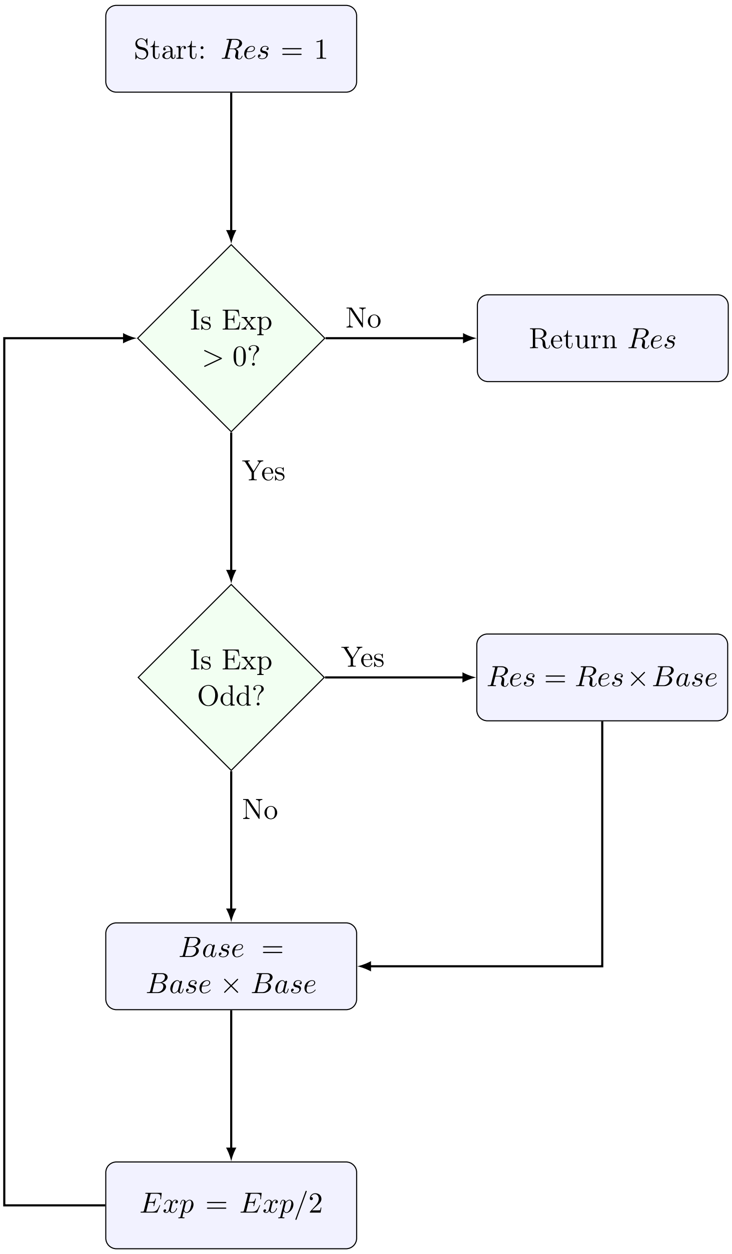

Figure 4: Binary Exponentiation Logic (Square-and-Multiply)

Chosen approach: Binary Exponentiation. It is significantly more efficient than naive iteration. Negative exponents are treated as integer reciprocals: since integer division truncates toward zero, $a^{-n} = 1/a^n$ evaluates to 0 for all $|a| > 1$. The result is non-zero (± 1) only if the base is ± 1.

Factorial Computing $n!$ presents these options:

- **Naive Multiplication:** Iteratively multiply $1 \times 2 \times \cdots \times n$. Simple but performs many multiplications of growing size.
- **Recursive Divide-and-Conquer:** Recursively split the range $[1, n]$ into halves, compute

their products, and multiply. This can balance operand sizes better than naive iteration.

- **Prime Factorization (Legendre's Formula):** Determine the prime factorization of $n!$ and compute the value by multiplying prime powers. This is efficient but complex.
- **Precomputed Table + Incremental Multiplication:** Store selected factorials (e.g., $10!, 20!, \dots$) in a table. To compute $n!$, load the closest precomputed value $\leq n$ and multiply upwards.

Chosen approach: Precomputed Table + Incremental Multiplication. This method provides a significant speedup for common input ranges by skipping the initial small multiplications, while remaining much simpler to implement than prime swing or full divide-and-conquer algorithms.

2.5 Parsing and evaluation model

Parsing must address operator precedence, associativity, and unary/postfix operators. The two standard approaches are Recursive-Descent Parsing and the Shunting-Yard Algorithm.

Chosen approach: Tokenizer + Shunting-Yard + Postfix Evaluator.

- **Separation of concerns:** Distinct phases for tokenization, reordering, and evaluation.
- **Uniformity:** Dijkstra's algorithm handles precedence and associativity uniformly.
- **Unary handling:** Complexities arise with the unary minus operator (e.g., in expressions like $-49!$ vs $(-49)!$). The solution is to handle this at the tokenization level: the tokenizer detects if a minus sign appears in a unary context (start of expression or after an operator) and emits a special, high-precedence internal operator ~. This ensures correct binding without the need for complex string rewriting.

2.6 Input/output semantics and two's-complement

Rationale for two's-complement: Machine integers naturally use two's-complement. Adopting this for binary/hex literals allows users to input bit patterns directly. Printing uses a "minimal two's-complement" format: positive numbers are prefixed with a zero bit/nibble if the MSB would otherwise imply a negative value, while negative numbers are printed in their standard two's-complement bit representation.

3 Implementation

This section documents the implementation details, data structures, and the public interface. The core logic resides in the `mp_int` module, with supporting modules for parsing, printing, and specific mathematical functions.

3.1 Design overview

The core structure is defined as follows:

```
typedef struct {
    int sign; /* 0 for zero, +1 or -1 for non-zero */
    mp_limb_t *digits; /* little-endian limb array (LSB first) */
    size_t length; /* number of used limbs */
    size_t capacity; /* allocated limb capacity */
} mp_int;
```

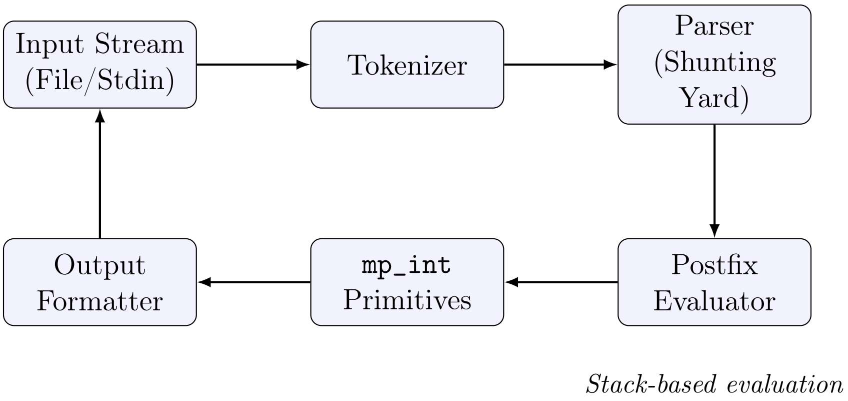

Figure 5: Logical Data Flow of the Calculator

Limb size is configured at compile time in `mp_int.h` to ensure that the product of two limbs fits into an `mp_double_t` accumulator:

- **64-bit systems (LP64):** `mp_limb_t` is `unsigned int` (32-bit), `mp_double_t` is `unsigned long` (64-bit).
- **32-bit systems:** `mp_limb_t` is `unsigned short` (16-bit), `mp_double_t` is `unsigned long` (32-bit).

3.2 Memory management

The module provides `mp_init`, `mp_free`, and `mp_reserve`.

- `mp_init` zeroes the structure.
- `mp_reserve` ensures sufficient capacity using `realloc`, handling OOM scenarios by returning `FAILURE`.
- `mp_free` releases the `digits` array and resets the structure.

3.3 Arithmetic primitives

The implementation includes internal helpers for small operand optimization:

- `mp_add_small`, `mp_mul_small`, `mp_div_small`: Optimized routines that operate on a single limb or scalar, avoiding full-precision overhead when possible.
- `mp_cmp_abs`, `mp_add_abs`, `mp_sub_abs`: Helpers for absolute value arithmetic, used as building blocks for signed operations.

3.4 Signed addition and subtraction

`mp_add` and `mp_sub` function as wrappers around the unsigned helpers.

- **Addition:** If signs match, magnitudes are added (`mp_add_abs`). If signs differ, the smaller magnitude is subtracted from the larger (`mp_sub_abs`), and the result takes the sign of the larger operand.
- **Subtraction:** Implemented by negating the second operand and calling `mp_add`.

3.5 Multiplication

The multiplication logic (`mp_mul`) selects an algorithm based on operand size:

1. **Naive Multiplication:** If either operand has fewer limbs than `NAIVE_THRESHOLD` (defined as 16), `mp_mul_naive` is used. This iterates through limbs using a double-width accumulator to propagate carries correctly.
2. **Karatsuba Multiplication:** For larger operands, `mp_karatsuba_mul` is invoked. It splits the inputs A and B into high and low halves (x_1, x_0 and y_1, y_0) and computes the product using three recursive multiplications:

   $$Z_0 = x_0 \cdot y_0, \quad Z_2 = x_1 \cdot y_1, \quad Z_1 = (x_0 + x_1)(y_0 + y_1) - Z_0 - Z_2$$

   The results are combined with appropriate bit-shifts.

3.6 Division and modulus

The division function `mp_div` (and similarly `mp_mod`) implements a "Shift and Subtract" algorithm using repeated doubling:

1. **Alignment:** The divisor is shifted left (word-wise) until it aligns with the most significant limbs of the dividend (remainder).
2. **Subtraction Loop:** While the remainder is greater than or equal to the original divisor:
   - A temporary candidate is initialized as the divisor.
   - The candidate is repeatedly doubled (left-shifted) as long as it remains smaller than the remainder.
   - This doubled candidate is subtracted from the remainder, and the corresponding bit/multiplier is added to the quotient.
3. **Shift Down:** The working divisor is shifted right by one word, and the process repeats until full precision is resolved.

This method avoids the complexity of quotient digit estimation found in Knuth's Algorithm D, favoring code simplicity and verification.

3.7 Exponentiation and factorial

Exponentiation: `mp_pow` utilizes the binary exponentiation (square-and-multiply) algorithm. It iterates through the bits of the exponent; for every bit set to 1, the base is multiplied into the result, and the base is squared at every step. Negative exponents trigger a check: if the result magnitude is 1, it returns ±1; otherwise, it returns 0 (integer division $1/N$).

Factorial: `mp_fact` avoids purely iterative multiplication by using a static table `fact_table` containing precomputed factorials up to 256!. The function identifies the largest precomputed $N! \leq A!$, initializes the accumulator with this value, and iteratively multiplies by integers from $N + 1$ up to A.

3.8 Parsing and expression evaluation

The parsing module uses a Shunting-Yard algorithm:

1. **Tokenization (`tokenize`):** The input string is split into tokens. The tokenizer detects whether a '+' or '-' character is a binary operator or a unary sign based on the context

(e.g., if it appears at the start of the expression or after an open parenthesis). If a unary minus is detected, it is emitted as a special ~ token.

2. **Shunting-Yard (`to_postfix`):** Infix tokens are converted to Reverse Polish Notation (postfix). The ~ operator is handled as a right-associative operator with higher precedence than addition/subtraction but lower than the factorial, ensuring correct binding (e.g., $-x! \to -(x!)$).
3. **Evaluation (`eval_postfix`):** The postfix queue is evaluated using a stack of `mp_int` values. The ~ operator pops one operand and negates it.

3.9 Printing semantics

Printing functions support three formats:

- **Decimal:** Implemented by repeatedly dividing the number by 10^9. The remainders are stored in a buffer and printed in reverse order as 9-digit zero-padded chunks.
- **Binary:** Prints the raw bits. For positive numbers, a leading `0` is added if the MSB is 1 (to denote positivity). For negative numbers, the value is converted to its two's-complement representation ($2^W - |X|$) and printed.
- **Hexadecimal:** Similar to binary but nibble-based. It calculates the minimal number of nibbles required and prints the two's-complement representation for negative values.

3.10 Runtime error reporting

Errors such as division by zero or invalid negative factorial inputs are handled via a global flag in `error.c`. The functions `calc_error_set()` and `calc_error_was_set()` allow the REPL to detect these conditions without crashing, printing a specific error message instead.

4 User manual

This section provides build instructions, usage examples, and a brief description of the command-line interface.

4.1 Compiling

Two Makefiles are provided: a standard Unix-style Makefile and a Windows-compatible variant.

Makefile (Unix / POSIX)

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi -Wextra -O2 -Iinclude -pg
BIN = calc.exe
...
```

Makefile.win (Windows / MinGW)

```
# Makefile.win -- for GNU make on Windows (MinGW/MSYS or Git-Bash)
CC ?= gcc
CFLAGS ?= -Wall -pedantic -ansi -Wextra -O2 -Iinclude -pg
...
```

To compile, run `make` in the project's base directory. No arguments are needed.

4.2 Command-line usage

Run without arguments to enter the interactive REPL:

```
$ ./calc.exe
> 2 + 2
4
> 5!
120
> hex
hex
> 255
0xff
> out
hex
> (0b111001010001 - 0x15ab)/ -(56+0b011010) + 10!
0x375f48
> dec
dec
> 2+2 +2 + 2+ -2
6
> 156! %0x35
0
> bin
bin
> 0x0FFF + 0x0FFF000
0b0111111111111111111111111
> quit
```

Run with a filename argument to process a script file line-by-line. The processor echoes each input line prefixed with > before printing the result. Example usage:

```
$ ./calc.exe input.txt
> 0b101011011101110100101011100011* -1573840394392989889898343/ 0x0fee
1329557753287977792226471797318
> 0xafadf7868373875afedbcddcbad ^ (-0b101)
-105554551620634528515688894704965755104871059409974374671373613377604305927069181300945998887
> 0x0fadf7868373875afedbcddcbad ^ (-0b0101)
0
> bin
bin
> (-9858756152423215485 5 % 0x829874ab3ff7398374fedbacdbacc -0b01011) ^ (0b011 % 0x37468237fcd
0b1011001110101100010101010000100001111110100001110111000110101110010101011110010010101110101
> hex
hex
> -(0xf1!^0b0100101)
Input of factorial must not be negative!
> -((-0xf1)!^0b0111)
0xca255c1f6ecaad7f1c7a66019f01695f9796ce9cc7a3572f7a3a0000000000000000000
> -49!^9
0xde35af7f5acc4c23f295d9d166bf49d25066f992282373231867c7595bd8d2944cea09acce2abfea948244a3aef
```

5 Conclusion

The project successfully delivers a functional and robust Multiple-Precision Integer Calculator compliant with ANSI C90 standards. The solution meets all core objectives, including arbitrary precision arithmetic, correct handling of signed inputs across decimal, binary, and hexadecimal bases, and a REPL allowing identical interactions with the user to the ones shown in the assignment.

The architectural choices—specifically the use of a limb-based representation and a hybrid multiplication strategy (combining Naive and Karatsuba algorithms)—provide a balanced trade-off between implementation complexity and runtime performance. The Shunting-Yard parser effectively handles operator precedence, while the tokenizer ensures correct interpretation of unary operators like the factorial.

Extensive testing confirms that the calculator behaves correctly across edge cases, such as two's-complement negative inputs, negative factorials, and zero-value operations. Memory management is handled explicitly, ensuring no leaks occur during operation. While the current division implementation is robust, future optimizations could include implementing Knuth's Algorithm D to improve performance on very large inputs. Factorial computation could be improved as well by implementing Luschny's Prime-Swing algorithm. Overall, the resulting software is stable, portable, and extensible.

6 Annex

6.1 Example input file used in 4.2 (Command-line usage)

`input.txt`

```
0b1010110111011101001010111000011* -157384039439298988989898343/ 0x0fee
0xafadf7868373875afedbcddcbad ^ (-0b101)
0x0fadf7868373875afedbcddcbad ^ (-0b0101)
bin
(-98587561524232154855 % 0x829874ab3ff7398374fedbacdbacc -0b01011) ^ (0b011 % 0x37468237fcdaabbcc)
hex
-(0xf1!^0b0100101)
-((-0xf1)!^0b0111)
-49!^9
```