PROJEKT KALKULATOR
Programowanie 1 - strukturalne

Mateusz Kozłowski

November 2023

Contents

1 Opis Projektu

Projekt zakłada stworzenie zaawansowanego kalkulatora matematycznego, który jest w stanie pracować z liczbami znacznie wykraczających poza limity standardowych typów danych nie tracąc przy tym precyzji obliczeń. W informatyce "Arbitrary-precision arithmetic", zwana także "Bignum arithmetic" wskazuje, że obliczenia są wykonywane na liczbach, których cyfry precyzji są ograniczone jedynie dostępną pamięcią systemu hosta. Kalkulator ten będzie zdolny do wykonywania podstawowych operacji matematycznych, takich jak dodawanie, odejmowanie, mnożenie, dzielenie, potęgowanie, operacji modulo, a także konwersji pomiędzy systemami liczbowymi, a wszystko to na liczbach liczbie cyfr przekraczającej setki tysięcy. Kalkulator BigNum może być wykorzystywany w różnych dziedzinach, w których precyzyjne obliczenia matematyczne są kluczowe. Przykładowo, gdy operujemy na dużych liczbach w naukach ścisłych, finansach, inżynierii lub kryptografii, standardowe typy danych mogą prowadzić do utraty dokładności lub nawet niemożności przeprowadzenia obliczeń. W takich przypadkach arytmetyka BigNum jest niezbędna do obsługi tych zadań.

1.1 Specyfikacja zadania

Napisać kalkulator działający na liczbach całkowitych o dowolnej wielkości (ograniczonych jedynie przez dostępną pamięć), na potrzeby szybkości wykonywania przyjmujemy argumenty o maksymalnej długości 40 cyfr oraz dla wykładnika 5 cyfr. Powinny zostać zdefiniowane operacje arytmetyczne z przypisanymi im operatorami:

- dodawania ' + '
- odejmowanie ' - '
- mnożenie ' * '
- dzielenie całkowitoliczbowe ' / '
- potęgowanie ' ^ '
- dzielenie modula ' % '
- konwersja systemu liczbowego ' [2 - 16] '

Program powinien zostać napisany w języku C, być uruchamiany w linii poleceń konsoli z pliku wykonywalnego *.exe oraz działać w systemie Windows.
Program wczytuje dane z pliku wejściowego i zapisuje wyniki w pliku wyjściowym. Nazwy plików podawane są jako argumenty wywołania. Jeśli nazwa pliku wyjściowego nie zostanie podana to program stworzy ją na podstawie nazwy pliku wejściowego:

Format nazwy pliku wejściowego $\Longrightarrow$ *.txt
Format nazwy pliku wyjściowego $\Longrightarrow$ out_*.txt

Uruchomienie:

```
$ : calc.exe < ścieżka do pliku wejściowego > [ ścieżka do pliku wyjściowego ]
```

Plik wejściowy:
Tekstowy plik wejściowy zawiera listę działań arytmetycznych, które program ma wykonać. Działania arytmetyczne zapisane są w formacie: znak działania, spacja, podstawa systemu argumentów, lista argumentów oddzielonych pustymi liniami.

```
< znak działania > < podstawa systemu argumentów >

<argument 1>

<argument 2>

<argument 3>

      ⋮
```

W przypadku konwersji w linii działania zapisane są: podstawa systemu argumentów, spacja, podstawa systemu docelowego. Poszczególne działania rozdzielone są trzema liniami przerwy.

```
< podstawa systemu argumentów > < podstawa systemu docelowego >

<argument 1>

      ⋮
```

Plik wyjściowy:
W pliku wyjściowym zapisywane są podane na wejściu działania wraz z wyliczonym wynikiem.

```
+ 16
A3
10
B3 // ⟸ wynik powyższych obliczeń

% 2
01000011
10101
100 // ⟸ wynik

10 16 // ⟸ konwersja systemu liczbowego 10 → 16
980
3D4 // ⟸ wynik konwersji systemy liczbowego
```

Program powinien odpowiednio radzić sobie z błędami, np. z błędnym formatem pliku wejściowym, czy błędnymi danymi w tym pliku. Powinien np. wykonać tyle działań, na ile jest to możliwe.

1.2 Wykorzystane oprogramowanie oraz technologie

Do stworzenia projektu został wykorzystany osprzęt:

- **Procesor:** AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
- **Zainstalowana pamięć RAM:** 16,0 GB (dostępne: 15,4 GB)
- **Typ systemu:** 64-bitowy system operacyjny, procesor x64
- **OS:** Windows 10 Home 22H2

Oprogramowanie:

- **Edytor kodu:** Visual Studio Code
- **Kompilator kodu:** GNU GCC
- **Online kompilator LaTeX:** Overleaf
- **System kontroli wersji:** Git

1.3 Uruchamianie i kompilacja projektu

Zakładamy, że :

- Użytkownik posiada zainstalowany kompilator gcc.exe, preferowana wersja to (MinGW.org GCC-6.3.0-1) 6.3.0 lub nowsza.
- Kompilator dodany do **"Environment Variables" - system PATH**
- Wiersz poleceń uruchamiany jest z katalogu, w którym znajdują się pliki źródłowe. Alternatywnie, jeśli użytkownik nie jest w katalogu, podaje pełną ścieżkę dostępu do plików.

Kompilacja calc.exe:

```
$ : gcc -std=c11 -Wall -Wextra -Werror -xc -lm vector.c bignum.c calculator.c main.c -o calc
```

Uruchomienie:

```
$ : ./calc < ścieżka do pliku wejściowego > [ ścieżka do pliku wyjściowego ]
```

Istnieje również możliwość przetestowania osobno poszczególnych funkcjonalności.
Kompilacja test.exe:

```
$ : gcc -std=c11 -Wall -Wextra -Werror -xc -lm vector.c bignum.c test.c -o test
```

Uruchomienie:

```
$ : ./test
```

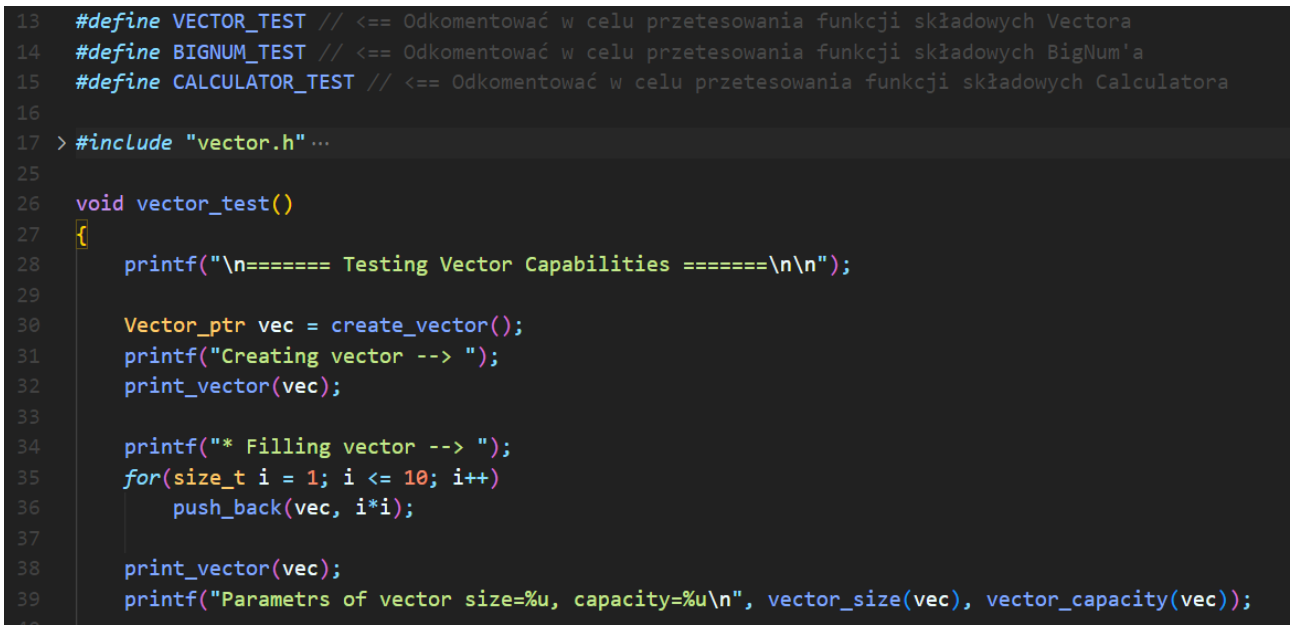

Fig. 1: W tym celu należy odkomentować/zakomentować poszczególne #define w pliku test.c

2 Organizacja kodu

W trakcie projektowania kalkulatora operującego na liczbach o nieokreślonej wielkości, postawiono sobie za cel wprowadzenie strukturyzacji kodu. Nie tylko miało to umożliwić efektywną realizację bieżących zadań, lecz także ułatwić przyszły rozwój, utrzymanie projektu, oraz umożliwić wykorzystanie aktualnie stworzonych modułów w zupełnie innych kontekstach.

Organizacja kodu opiera się na wykorzystaniu dwóch kluczowych struktur: 'Vector' i 'BigNum', z których każda pełni klarowną rolę w kontekście operacji wykonywanych w 'Calculator'. Struktura 'Vector' jest dynamiczną tablicą, dostosowującą automatycznie swój rozmiar w trakcie działania programu. Oferuje szereg funkcji do manipulowania danymi, takich jak dodawanie, usuwanie czy modyfikowanie istniejących już elementów. Przypomina kontenery danych dostępne w innych nowoczesnych językach programowania, np. w C++.
Z kolei 'BigNum' bazując na strukturze 'Vector', rozszerza ją o dodatkowe pola sign oraz num_system stworząc stukturę, którą możemy interpretować jako liczba. Zdefiniowane zostały dla niej funkcjonalność do obsługi arytmetyki dużych liczb - operacje dodawania, odejmowania, mnożenia ... oraz konwersji na inny system liczbowy.

Omówiona w następnej sekcji struktura 'Calculator' łączy możliwości struktury 'BigNum' z obsługą plików tekstowych - tym samym tworząc w pełni funkcjonalny program.

2.1 Vector

Vectory to kontenery sekwencyjne reprezentujące tablice, które mogą w czasie wykonywania programu zmieniać swoją wielkość.

Podobnie jak tablice, vectory używają ciągłych lokalizacji pamięci do przechowywania swoich elementów, co oznacza, że ich elementy mogą być dostępne za pomocą przesunięć na zwykłych wskaźnikach do ich elementów, i równie efektywnie jak w tablicach. Jednak w przeciwieństwie do tablic, ich rozmiar może zmieniać się dynamicznie, a ich pamięć jest obsługiwana automatycznie przez kontener.

Wewnętrznie wektory używają dynamicznie alokowanej tablicy do przechowywania swoich

elementów. Ta tablica może być ponownie alokowana, aby zwiększyć rozmiar, gdy dodawane są nowe elementy, co oznacza alokację nowej tablicy i przenoszenie wszystkich elementów do niej. Jest to zadanie stosunkowo kosztowne pod względem czasu przetwarzania, dlatego wektory nie realokują pamięci za każdym razem, gdy dodawany jest element do kontenera.

Zamiast tego kontenery wektorów mogą alokować dodatkową pamięć, aby pomieścić ewentualny wzrost, co oznacza, że kontener może mieć rzeczywistą pojemność większą niż pamięć potrzebna do przechowywania jego elementów (tj. jego rozmiar). Biblioteki mogą stosować różne strategie wzrostu w celu zbalansowania między użyciem pamięci a realokacjami, ale w każdym przypadku realokacje powinny występować tylko w logarytmicznie rosnących odstępach wielkości, aby wstawienie pojedynczych elementów na koniec wektora mogło być zapewnione z amortyzowanym stałym czasem złożoności. W naszym przypadku jest to dwukrotne zwiększenie rozmiaru vectora.

```c
/* Type definition for the elements stored in the vector */
#define DATA_TYPE unsigned char

/* Type definition for a NULL value in the vector */
#define NULL_TYPE NULL

/* Structure to represent a dynamic array (vector) */
typedef struct {
    size_t size;      /* Number of elements in the vector */
    size_t capacity;  /* Current capacity of the vector */
    DATA_TYPE *array; /* Pointer to the dynamic array */
} Vector;

/* Defining Vector_ptr as a pointer to a Vector structure */
typedef Vector *Vector_ptr;
```

Fig. 2: Implementacja struktury Vector

2.1.1 Funkcje obsługujące Vector

```
Function: Vector_ptr create_vector();
-------------------------------------
Allocates memory for a new vector and initializes its properties.

Parameters: None

Returns:
- A pointer to the newly created vector on success.
- NULL_TYPE if memory allocation fails during vector creation.

*****************************************************************************
Function: Vector_ptr delete_vector(Vector_ptr ptr);
---------------------------------------------------
Deletes a vector and frees the associated memory.

Parameters:
- ptr: A pointer to the vector to be deleted.

Returns:
- NULL_TYPE after freeing the vector's memory.

*****************************************************************************
Function: Vector_ptr copy_vector(Vector_ptr ptr);
-------------------------------------------------
Creates a new vector, resizes it, and copies elements from the input vector.

Parameters:
- ptr: A pointer to the vector to be copied.

Returns:
- A pointer to the newly created vector on success.
- NULL_TYPE if the copy operation fails due to memory allocation issues.

*****************************************************************************
Function: Vector_ptr reverse_vector(Vector_ptr ptr);
----------------------------------------------------
Reverses the order of elements in the vector.

Parameters:
- ptr: A pointer to the vector to be reversed.

Returns:
- A pointer to the reversed vector on success.
- NULL_TYPE if the input vector is invalid.
```

```
*****************************************************************************
Function: void vector_reserve(Vector_ptr ptr, size_t new_capacity);
-------------------------------------------------------------
Resizes the vector to a new specified capacity. If memory allocation success,
vector capacity is changed, otherwise vector remains unchanged.

Parameters:
- ptr: A pointer to the vector to be resized.
- new_capacity: The new capacity for the vector.

Returns: None

*****************************************************************************
Function: vector_resize
----------------------
Resizes the container so that it contains new_size elements.
If new_size < size, the content is reduced to its first n elements, removing those beyon
If new_size > size, the content is expanded by inserting at the end as many elements as
If new_size > capacity, an automatic reallocation of the allocated storage space takes p

Parameters:
- ptr: A pointer to the vector to be resized.
- new_size: The new size for the vector.

*****************************************************************************
Function: void vector_clear(Vector_ptr ptr);
----------------------------------------
Removes all elements from the vector while keeping the capacity unchanged.

Parameters:
- ptr: A pointer to the vector to be cleared.

Returns: None

*****************************************************************************
Function: void push_back(Vector_ptr ptr, DATA_TYPE val);
-----------------------------------------------------
Adds an element to the end of the vector. Doubles capacity if the vector is full.
Calls vector_resize function, if operation faills push_back isn't performed.

Parameters:
- ptr: A pointer to the vector.
- val: The value to be added to the vector.

Returns: None

*****************************************************************************
Function: void pop_back(Vector_ptr ptr);
---------------------------------------
Removes the last element from the vector.
```

```
Parameters:
- ptr: A pointer to the vector.

Returns: None

*****************************************************************************
Function: DATA_TYPE vector_front(Vector_ptr ptr);
-------------------------------------------------
Gets the value of the first element in the vector.

Parameters:
- ptr: A pointer to the vector.

Returns:
- The value of the first element.
- -1 if the vector is empty or invalid.

*****************************************************************************
Function: DATA_TYPE vector_back(Vector_ptr ptr);
------------------------------------------------
Gets the value of the last element in the vector.

Parameters:
- ptr: A pointer to the vector.

Returns:
- The value of the last element.
- -1 if the vector is empty or invalid.

*****************************************************************************
Function: DATA_TYPE vector_get(Vector_ptr ptr, size_t index);
-------------------------------------------------------------
Gets the value of an element at a specific index in the vector.

Parameters:
- ptr: A pointer to the vector.
- index: The index of the element to retrieve.

Returns:
- The value of the element at the specified index.
- -1 if the vector is invalid or the index is out of bounds.

*****************************************************************************
Function: void vector_set(Vector_ptr ptr, size_t index, DATA_TYPE val);
-----------------------------------------------------------------------
Sets the value of an element at a specific index in the vector.

Parameters:
- ptr: A pointer to the vector.
```

```
- index: The index of the element to be set.
- val: The value to set.

Returns: None

*****************************************************************************
Function: size_t vector_size(Vector_ptr ptr);
-----------------------------------------
Gets the current number of elements in the vector.

Parameters:
- ptr: A pointer to the vector.

Returns:
- The current number of elements in the vector.
- 0 if the vector is invalid.

*****************************************************************************
Function: size_t vector_capacity(Vector_ptr ptr);
-------------------------------------------------
Gets the current capacity of the vector.

Parameters:
- ptr: A pointer to the vector.

Returns:
- The current capacity of the vector.
- 0 if the vector is invalid.

*****************************************************************************
Function: bool vector_empty(Vector_ptr ptr);
--------------------------------------------
Checks if the vector is empty (contains no elements).

Parameters:
- ptr: A pointer to the vector.

Returns:
- true if the vector is empty or invalid, false otherwise

*****************************************************************************
Function: void print_vector(Vector_ptr ptr);
--------------------------------------------
Prints the elements of the vector to the standard output

Parameters:
- ptr: A pointer to the vector.

Returns: None
```

2.1.2 Przykładowe użycie struktury Vector

```c
void vector_test()
{
    printf("\n======= Testing Vector Capabilities =======\n\n");

    Vector_ptr vec = create_vector();
    printf("Creating vector --> ");
    print_vector(vec);

    printf("* Filling vector --> ");
    for(size_t i = 1; i <= 10; i++)
        push_back(vec, i*i);

    print_vector(vec);
    printf("Parametrs of vector size=%u, capacity=%u\n", vector_size(vec), vector_capacity(vec));

    printf("* Reversed vector --> ");
    reverse_vector(vec);
    print_vector(vec);

    pop_back(vec);
    vector_set(vec, 5, 13);
    printf("* Setting vec[5]= %u and removing last element -->", vector_get(vec, 5));
    print_vector(vec);

    vec = delete_vector(vec);
}
```

Fig. 3: Dokładnie przetestowana struktura Vector dostępna do wglądu w pliku test.c

2.2 BigNum

BigNum jest to typ danych używany w programowaniu do reprezentowania liczb całkowitych o bardzo dużych wartościach, które znacznie przekraczają zakres standardowych typów danych całkowitych, takich jak int czy long long. Pozwala na wykonywanie operacji na liczbach całkowitych o dowolnej wielkości, ograniczonej jedynie dostępną pamięcią komputera.

Zdefiniowana została dla niego arytmetyka dużych liczb - "Arbitrary-precision arithmetic". Dostępne operacje na BigNum to dodawanie, odejmowanie, mnożenie, dzielenie całkowitoliczbowe, potęgowanie, operacja modulo oraz konwersja pomiędzy systemami liczbowymi. Zaimplementowana jest na bazie Vectora, gdzie każda cyfra przechowywana jest jako oddzielny element dynamicznej tablicy.

```c
#define PLUS 1      /* Positive sign bit */
#define MINUS -1    /* Negative sign bit */

/* BigNum implementation is based on the vector container */
typedef struct {
    int sign_bit;        /* Sign bit (PLUS or MINUS) */
    Vector_ptr digits;   /* Vector for storing the digits of the number in Little-Endian system*/
    size_t num_system;   /* Number system (e.g., decimal, binary) */
} BigNum;

/* Defining BigNum_ptr as a pointer to a BigNum structure */
typedef BigNum *BigNum_ptr;
```

Fig. 4: Implementacja struktury BigNum

2.2.1 Funkcje obsługujące BigNum

```
Macro: MIN(X, Y) (((X) < (Y)) ? (X) : (Y))
---------------
Returns the minimum value between X and Y.
If X is less than Y, it evaluates to X; otherwise, it evaluates to Y.

*****************************************************************************
Macro: MAN(X, Y) (((X) > (Y)) ? (X) : (Y))
---------------
Returns the maximum value between X and Y.
If X is greater than Y, it evaluates to X; otherwise, it evaluates to Y.

*****************************************************************************
Function: BigNum_ptr create_BigNum();
------------------------
Allocates memory for a new BigNum and initializes its properties.

Returns:
- A pointer to the newly created BigNum on success.
- NULL_TYPE if memory allocation fails during BigNum creation.

*****************************************************************************
Function: BigNum_ptr delete_BigNum(BigNum_ptr bg);
------------------------
Deletes a BigNum and frees the associated memory.

Parameters:
- bg: A pointer to the BigNum to be deleted.

Returns:
- NULL_TYPE after freeing the BigNum's memory.

*****************************************************************************
Function: BigNum_ptr assign_value(BigNum_ptr bg, const char *num_cstring, size_t system)
----------------------
Assigns a new value to an existing BigNum based on a C string representation.

Parameters:
- bg: A pointer to the target BigNum.
- num_cstring: C string representation of the number.
- system: The number system in which the input is represented.

Returns:
- A pointer to the updated BigNum on success.
- NULL_TYPE if there is an error, such as invalid input or memory allocation failure.

*****************************************************************************
Function: BigNum_ptr int_to_BigNum(int val);
------------------------
Creates a new BigNum from an integer value.
```

```
Number system set to decimal by default.

Parameters:
- val: The integer value.

Returns:
- A pointer to the newly created BigNum on success.
- NULL_TYPE if there is an error, such as memory allocation failure.

*****************************************************************************
Function: BigNum_ptr copy_BigNum(BigNum_ptr bg);
----------------------
Creates a new BigNum that is a copy of the provided BigNum.

Parameters:
- bg: A pointer to the source BigNum.

Returns:
- A pointer to the newly created BigNum (copy) on success.
- NULL_TYPE if there is an error, such as memory allocation failure.

*****************************************************************************
Function: void clear_BigNum(BigNum_ptr bg);
----------------------
Resets the value of a BigNum to NaN, usesy clear_vector.

Parameters:
- bg: A pointer to the target BigNum.

*****************************************************************************
Function: int map_digit(unsigned char digit);
--------------------
Maps an ASCII character representing a digit to its corresponding integer value.

Parameters:
- digit: The ASCII character representing a digit.

Returns:
- The integer value of the digit.

*****************************************************************************
Function: int BigNum_sign(BigNum_ptr bg);
--------------------
Gets the sign bit of a BigNum.

Parameters:
- bg: A pointer to the BigNum.

Returns:
- The sign bit (PLUS or MINUS).
```

```
**************************************************************************
Function: size_t BigNum_size(BigNum_ptr bg);
---------------------
Gets the number of digits in a BigNum.

Parameters:
- bg: A pointer to the BigNum.

Returns:
- The number of digits in the BigNum

**************************************************************************
Function: size_t BigNum_base(BigNum_ptr bg);
---------------------
Gets the number system (base) of a BigNum.

Parameters:
- bg: A pointer to the BigNum.

Returns:
- The number system (base) of the BigNum.

**************************************************************************
Function: void zero_justify(BigNum_ptr bg);
-----------------------
Removes leading zero digits from a BigNum.

Parameters:
- bg: A pointer to the target BigNum.

**************************************************************************
Function: BigNum_ptr add_BigNum(BigNum_ptr bg1, BigNum_ptr bg2);
---------------------
Adds two BigNums and returns the result.

Parameters:
- bg1: A pointer to the first BigNum.
- bg2: A pointer to the second BigNum.

Returns:
- A pointer to the resulting BigNum on success.
- NULL_TYPE if there is an error, such as invalid input or memory allocation failure.

**************************************************************************
Function: BigNum_ptr subtract_BigNum(BigNum_ptr bg1, BigNum_ptr bg2);
-------------------------
Subtracts the second BigNum from the first and returns the result.

Parameters:
```

```
- bg1: A pointer to the first BigNum.
- bg2: A pointer to the second BigNum.

Returns:
- A pointer to the resulting BigNum on success.
- NULL_TYPE if there is an error, such as invalid input or memory allocation failure.

*****************************************************************************
Function: BigNum_ptr multiply_BigNum(BigNum_ptr bg1, BigNum_ptr bg2);
-------------------------
Multiplies two BigNums and returns the result.

Parameters:
- bg1: A pointer to the first BigNum.
- bg2: A pointer to the second BigNum.

Returns:
- A pointer to the resulting BigNum on success.
- NULL_TYPE if there is an error, such as invalid input or memory allocation failure.

*****************************************************************************
Function: BigNum_ptr divide_BigNum(BigNum_ptr bg1, BigNum_ptr bg2);
-----------------------
Divides the first BigNum by the second and returns the result.

Parameters:
- bg1: A pointer to the dividend BigNum.
- bg2: A pointer to the divisor BigNum.

Returns:
- A pointer to the resulting BigNum on success.
- NULL_TYPE if there is an error, such as division by zero or memory allocation failure.

*****************************************************************************
Function: BigNum_ptr exponentiate_BigNum(BigNum_ptr bg1, BigNum_ptr bg2);
-----------------------------
Exponentiates the first BigNum by the second and returns the result.

Parameters:
- bg1: A pointer to the base BigNum.
- bg2: A pointer to the exponent BigNum.

Returns:
- A pointer to the resulting BigNum on success.
- NULL_TYPE if there is an error, such as invalid input or memory allocation failure.

*****************************************************************************
Function: BigNum_ptr modulo_BigNum(BigNum_ptr bg1, BigNum_ptr bg2);
-----------------------
Computes the modulo of the first BigNum by the second and returns the result.
```

```
Parameters:
- bg1: A pointer to the dividend BigNum.
- bg2: A pointer to the divisor BigNum.

Returns:
- A pointer to the resulting BigNum on success.
- NULL_TYPE if there is an error, such as division by zero or memory allocation failure.

*****************************************************************************
Function: BigNum_ptr convert_system_BigNum(BigNum_ptr bg, size_t system);
------------------------------
Converts a BigNum to a different number system.

Parameters:
- bg: A pointer to the BigNum to be converted.
- system: The target number system.

Returns:
- A pointer to the converted BigNum on success.
- NULL_TYPE if there is an error, such as an unsupported number system or memory allocat

*****************************************************************************
Function: int compare_BigNum(BigNum_ptr bg1, BigNum_ptr bg2);
-------------------------
Compares two BigNums

Parameters:
- bg1: A pointer to the first BigNum.
- bg2: A pointer to the second BigNum.

Returns:
- -1 if bg1 < bg2.
- 0 if bg1 == bg2.
- 1 if bg1 > bg2.

*****************************************************************************
Function: int compare_abs_BigNum(BigNum_ptr bg1, BigNum_ptr bg2);
---------------------------
Compares the absolute values of two BigNums
Parameters:
- bg1: A pointer to the first BigNum.
- bg2: A pointer to the second BigNum.

Returns:
- -1 if |bg1| < |bg2|.
- 0 if |bg1| == |bg2|.
- 1 if |bg1| > |bg2|.

*****************************************************************************
```

```
Function: char *BigNum_to_cstring(BigNum_ptr bg);
---------------------------
Converts a BigNum to a C string representation.

Parameters:
- bg: A pointer to the BigNum to be converted.

Returns:
- A pointer to the C string representation on success.
- NULL_TYPE if there is an error, such as memory allocation failure.

*****************************************************************************
Function: void print_BigNum(BigNum_ptr bg);
----------------------
Prints the array representation of a BigNum to the standard output.

Parameters:
- bg: A pointer to the BigNum to be printed.
```

2.2.2 Przykładowe użycie struktury BigNum

```c
// Create a BigNum
BigNum_ptr bg1 = create_BigNum();
assign_value(bg1, "123", 9);
printf("Assigning value: bg1 = (123)[9]: ");
print_BigNum(bg1);

// Create another BigNum
BigNum_ptr bg2 = int_to_BigNum(-125); // Decimal system
convert_system_BigNum(bg2, 9);
printf("Assigning value: bg2 = -(125)[9]: ");
print_BigNum(bg2);

// Multiply two BigNums
multiply_BigNum(bg1, bg2);
printf("bg1 * bg2 = ");
print_BigNum(bg1);

// Assign a new value in hexadecimal
assign_value(bg1, "FB", 16);
printf("\n\nAssigned FB to BigNum (hexadecimal): ");
print_BigNum(bg1);

// Compare BigNums
assign_value(bg2, "-FB", 16);
printf("Created BigNum2 (hexadecimal): ");
print_BigNum(bg2);
int comparison = compare_BigNum(bg1, bg2);
printf("Comparison result (BigNum1 cmp BigNum2): %d\n", comparison);
printf("|ABS| comparison result ( |BigNum1| cmp |BigNum2|): %d\n", compare_abs_BigNum(bg1, bg2));

// Convert BigNum to a C string
char *str = BigNum_to_cstring(bg2);
printf("BigNum2: %s\n", str);
free(str);

// Cleanup
delete_BigNum(bg1);
delete_BigNum(bg2);
```

Fig. 5: Przykładowe wykorzystanie możliwości BigNum

3 Kalkulator

Struktura kalkulator wykorzystuje możliwości oferowane przez BigNum do wykonywania operacji arytmetycznych, dodatkowo umożliwia wykonywanie pracy na plikach tekstowych - odczytywania i zapisywania obliczeń. Wyłapuje podstawowe błędy takie jak błędne formatowanie pliku wejściowego czy niedozwolone operacje dając informacje na pliku wyjściowym o rodzaju błędu.

```c
enum Operation{
    ADD = 0,
    SUBSTRACT,
    MULTIPLY,
    DIVIDE,
    EXPONENTIATE,
    MODULO,
    CHANGE_BASE,
    UNDEFINED
};

/*...
enum Error_flag{
    VALID = 0,
    INVALID_OPERATOR,
    INVALID_BASE,
    INVALID_NUMBER_OF_ARG,
    INVALID_NUMBER,
    DIVISION_BY_ZERO
};

/* Represents a calculator with various properties for managing data */
typedef struct {
    BigNum_ptr memory;      // Memory for storing calculated results
    BigNum_ptr act_number;  // Active number for the current operation
    bool ready_result;      // Flag to indicate if the result is ready
    char *in_file;          // Input file name
    char *out_file;         // Output file name
    fpos_t index;           // File position index for reading input file
    enum Operation operation;   // Current operation
    enum Error_flag flag;       // Status flag for error handling
    size_t old_base;            // Base of the current number (for system conversion)
    size_t new_base;            // Target base for system conversion
} Calc;

/* Defining Calc_ptr as a pointer to a Calc structure*/
typedef Calc *Calc_ptr;
```

Fig. 6: Implementacja struktury Calc oraz zdefiniowanie stanów z użyciem enum

3.1 Funkcje obsługujące Kalkulator

```
Function: Calc_ptr create_calculator(char* input_file, char* output_file);
---------------------------
```

```
Allocates memory for a new calculator and initializes its properties.

Parameters:
- input_file: Name of the input file.
- output_file: Name of the output file.

Returns:
- A pointer to the newly created calculator on success.
- NULL_TYPE if memory allocation fails during calculator creation.

*****************************************************************************
Function: Calc_ptr delete_calculator(Calc_ptr ptr);
--------------------------
Deletes a calculator and frees the associated memory.

Parameters:
- ptr: A pointer to the calculator to be deleted.

Returns:
- NULL_TYPE after freeing the calculator's memory.

*****************************************************************************
Function: char **split_line(const char buffer[]);
--------------------
Splits a line of text into an array of words.

Parameters:
- buffer: A character array containing the input line.

Returns:
- A dynamically allocated array of strings representing words in the line.
- NULL if the line is empty.

*****************************************************************************
Function: char **load_line(char *input_file, fpos_t *pos);
-------------------
Loads a line from the input file at a specific position.

Parameters:
- input_file: Name of the input file.
- pos: Pointer to the file position index.

Returns:
- An array of strings representing the line on success.
- NULL if file opening fails or if the line cannot be loaded.

*****************************************************************************
Function: char **line_memory_deallocation(char **ptr);
-----------------------------------
Deallocates memory used by a line.
```

```
Parameters:
- ptr: A pointer to the array of strings representing a line.

Returns:
- NULL_TYPE after freeing the line's memory.

*****************************************************************************
Function: int words_in_line(char **line);
-----------------------
Counts the number of words in a line.

Parameters:
- line: An array of strings representing a line.

Returns:
- The number of words in the line.

*****************************************************************************
Function: void append_line(char *output_file, char *str);
---------------------
Appends a cstring to the output file.

Parameters:
- output_file: Name of the output file.
- str: The string to be appended to the file.

*****************************************************************************
Function: void save_line(Calc_ptr calc, char **line);
-------------------
Concatenates element's of line and save them ussing append_line(..) to the output file.

Parameters:
- calc: A pointer to the calculator structure.
- line: An array of strings representing a line.

*****************************************************************************
Function: void save_error(Calc_ptr calc);
--------------------
Saves an error message to the output file.

Parameters:
- calc: A pointer to the calculator structure.

*****************************************************************************
Function: bool is_base(const char *str);
-----------------
Checks if a string represents a valid base.

Parameters:
```

```
- str: The string to be checked.

Returns:
- true if the string represents a valid base, false otherwise.

*****************************************************************************
Function: bool is_num(const char *str, int num_base);
---------------
Checks if a string represents a valid number in a given base.

Parameters:
- str: The string to be checked.
- num_base: The base for the number.

Returns:
- true if the string represents a valid number, false otherwise.

*****************************************************************************
Function: void update_operation_status(Calc_ptr calc, char** line);
--------------------------------
Updates the calc->operation status based on the input line.

Parameters:
- calc: A pointer to the calculator structure.
- line: An array of strings representing a line.

*****************************************************************************
Function: void make_calculations(Calc_ptr, int *argument_counter);
---------------------------
Performs calculations based on the current operation.

Parameters:
- calc: A pointer to the calculator structure.
- argument_counter: A pointer to the counter for the number of arguments.

*****************************************************************************
Function: void start_calculations(Calc_ptr ptr);
---------------------------
Starts the calculator and performs calculations from the input file.
Used as a main program loop.

Parameters:
- ptr: A pointer to the calculator structure.
```

3.2 Opis obsługiwanych błędów

- **Kończące program:**
  - Niepoprawna ilość argumentów wiersza poleceń
  - Podanie nieistniejących nazw plików
- **(Pomijane) Sygnalizujące błąd w pliku wyjściowym:**
  - Niepoprawny operator $\Longrightarrow$ [err: INVALID_OPERATOR]
  - Niepoprawna baza $\Longrightarrow$ [err: INVALID_BASE]
  - Niepoprawna liczba argumentów $\Longrightarrow$ [err: INVALID_NUMBER_OF_ARG]
  - Liczba niepasująca do bazy $\Longrightarrow$ [err: INVALID_NUMBER]
  - Dzielenie przez zero $\Longrightarrow$ [err: DIVISION_BY_ZERO]

4 Podsumowanie i wnioski

Główne cechy projektu:

1. Struktura BigNum: Zaimplementowano strukturę BigNum, która umożliwia operacje na liczbach o dużej precyzji, obsługując różne systemy liczbowe.
2. Kalkulator: Stworzono kalkulator, który korzysta z funkcji dostarczonych przez strukturę BigNum do wykonywania podstawowych operacji matematycznych, takich jak dodawanie, odejmowanie, mnożenie, dzielenie, reszta z dzielenia, potęgowanie, oraz porównywanie liczb.
3. Wczytywanie i zapisywanie do plików: Projekt umożliwia użytkownikowi zapisywanie wyników obliczeń do plików oraz wczytywanie danych z plików, co zwiększa funkcjonalność kalkulatora.
4. Obsługa błędów: Dodano mechanizmy do wyłapywania i obsługi błędów, co poprawia bezpieczeństwo użytkowania kalkulatora. Użytkownik otrzymuje czytelne komunikaty o ewentualnych problemach.

Możliwe ulepszenia:

- Zastosowanie szybszych algorytmów mnożenia np:
  Karatsuba, Toom 3-Way, czy Toom 4-Way
- Technik Divide and Conquer w dzieleniu oraz potęgowaniu
- Pominięcie przejściowej konwersji do systemu dziesiętnego podczas zmiany bazy